Dominik Jürgens

Survey on Software Engineering for Scientific Applications – Reusable Software, Grid Computing and Application

Braunschweig : Inst. für Wissenschaftliches Rechnen, 2009

Informatikbericht / Technische Universität Braunschweig ; Nr. 2009-02

Veröffentlicht: 12.05.2009

http://www.digibib.tu-bs.de/?docid=00027815

Dominik Jürgens

Survey on Software Engineering for Scientific Applications

Reuseable Software, Grid Computing and Application

INSTITUTE OF SCIENTIFIC COMPUTING
CARL-FRIEDRICH-GAUSS-FAKULTÄT
TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG

Braunschweig, Germany, 2009

Survey on Software Engineering for Scientific Applications

Dominik Jürgens
Institute of Scientific Computing
Carl-Friedrich-Gauß-Fakultät
Technische Universität Braunschweig, Germany

Informatikbericht Nr.: 2009-02

Freitag, 13. März 2009

Location
Institute of Scientific Computing
Technische Universität Braunschweig
Hans-Sommer-Straße 65
D-38106 Braunschweig

Postal Address
Institut für Wissenschaftliches Rechnen
Technische Universität Braunschweig
D-38092 Braunschweig
Germany

Contect
Phone: +49-(0)531-391-3000
Fax: +49-(0)531-391-3003
E-Mail: wire@tu-bs.de
URL: http://www.wire.tu-bs.de

Abstract

Fields of modern science and engineering are in need of solving more and more complex numerical problems. The complexity of scientific software thereby rises continuously. This growth is caused by a number of changing requirements. Coupled phenomena gain importance and new technologies like computational-grids, graphical and heterogeneous multi-core processors have to be used to achieve high-performance.

The amount of additional complexity can not be handled by a small group of specialised scientists. The interdiciplinary nature of scientific software thereby presents new challanges for software engineering. A paradigm shift towards a stronger separation of concerns becomes necessary in the development of future scientific software.

The coupling of independently simulated physical phenomena is an important example for a software-engineering concern in the domain of computational science. In this context, different simulation-programs model only a part of a more complex *coupled* system. The present work gives overview on paradigms which aim at making software-development in computational sciences more reliable and less interdependent. A special focus is put on the development of coupled simulations.

Contents

List of Figures

Preface

An important scientific innovation rarely makes its way
by gradually winning over and converting its opponents:
it rarely happens that Saul becomes Paul.
What does happen is that the opponents gradually die out,
and that the growing generation is familiarised with the
ideas from the beginning.
MAX PLANCK, SCIENTIFIC AUTOBIOGRAPHY, 1949

I have chosen this preface as space for a few personal words about this work.

As graduate in computer science and doctoral student in computational science, my goal with this literature review is generally to win overview and to find space for my scientific contribution.

I started the review with a consideration of component based software engineering (CBSE). The first part of this review therefore shortly presents the evolution of the metaphor, *software-component*, from the first ideas towards the modern view. Application of CBSE in computational science was thereby always in my mind. But when I reviewed material from computational sciences I realised that the understanding of programming is different compared to computer sciences. While object-oriented modelling is an old hat in software engineering, in computational science it lives in the shadow of procedural programming and is often (if at all) used as a programming technique rather than a modelling framework. Scientific software written in C++ is often called object oriented, only because the C compiler would not accept the code. Software engineering, engineering processes, separation of concerns and object oriented modelling are often understood as overhead.

During the review, I additionally became interested into generative programming, so I extended the programming chapter with a short consideration of this context. Generative programming is also related to CBSE, but with a more general definition of components, so the consideration made sense to me.

Beside CBSE another important area in computational sciences is parallel computing. Parallel computing was an important topic in my studies; I have chosen subjects like “parallel system architecture” and “parallel computing”. Just at the time, when I started the review, a new discusson on parallel programming emerged. This new wave of interest into parallel programming was excited by a paradigm shift of the semi-conductor industry, which shifted from an *increase clockrate* to a *increase parallelism* policy. The review also takes some hardware-details into account which help for a better understanding of the onging discussions.

To combine these interests with computational science is not the hardest job, because of the following reason. At the institute of scientific computing — my employer — software engineering has tradition, it originates a scientific software component framework, the Component Template Library (CTL), which is the first generative component framework. Rainer Niekamp, *the master of the CTL*, one day came to my office and remarked, that it might be interesting for me to consider coupled simulations. In this research, software engineering with CBSE and abstractions for parallel programming are of great interest. I followed his advise and from that day on, I started to work myself into this field with the goal of applying all the things I am interested in. It turned out, that coupled simulation is a field in computational science where indeed all of my interests may be of great use. Thank you Rainer.

Caused by Rainers advice, I started the third part of this review: coupled simulation.

Chapter 1

Introduction

In this introductory chapter, the three main topics **Software Engineering in Computational Science**, **Grid Computing** and **Coupled Simulation** will be shortly introduced to motivate their consideration in this text.

Software Engineering in Computational Science

It is unworthy of excellent men to lose hours like slaves in the labor of calculation which could be relegated to anyone else if machines were used.
GOTTFRIED WILHELM VON LEIBNITZ (1646-1716)

General Remark

There are many different participants involved in the development of software which causes many different points of view on the same thing. The *user* of a software system is interested in usability, functionality and costs. A user does generally not think about implications of his needs. *Software-architects* see the customers requirements as the most critical part of the development and try to reduce degrees of freedom for the programmer. *Programmers* see technical problems and have to interact with the architect to find an adequate solution fulfilling the needs of the customer or scientific project. The *customer-service* of a commercial software-product only sees the problems which the customers have. The *manager* (or project-manager) is interested in the costs, which have to be reduced. The *controlling* sees the whole process and wants to optimise the quality of the product while reducing the costs. The list does not end here, but one can easily derive that there is much literature, many publications, and divorce opinions. In the development of non-commercial scientific software, all these aspects have to be managed without persons explicitly working for it. Often the project leader is user, software-architect, programmer, customer-service, manager and controller in one person, which hinders the application of many established approaches.

What is Different in Scientific Computing?

Today, computer simulation plays an important role in engineering and science. Scientific computing is an interdisciplinary field, where modelling of complex systems using numerical analysis and their efficient software implementation are important topics. Software for simulating complex phenomena has to be carefully designed to provide high-performance and suitable results in appropriate time. Often, special knowledge about hardware and software components has to be used to achieve these goal.

It is the combination of engineering, natural science, computer science and mathematics, making scientific computing a demanding field for all participating parties: Engineers contribute challenging applications and technical knowledge; physicists and other natural scientists contribute their models; mathematicians contribute numerical

methods and algorithms for the simulation of complex processes; computer scientists contribute infrastructure, data structures, algorithms and last but not least, programming languages.

Many scientific and engineering disciplines have great advantages from using numerical models to simulate complex scenarios [Fox02]. The industry uses virtual prototypes e.g. to accelerate product development, to shorten design phases and to optimise products with respect to otherwise hardly observable phenomena like e.g. turbulence, material vibration, etc. Climate science use numerical models to predict weather or to understand its past.

This indicates that computational science is a field in which experts from different scientific domains have to work closely together to solve challenging problems. In this demanding field the challenge of solving complex computational problems is inherent and therefore the reason why scientists have to use state-of-the-art technology and methods from many scientific fields. In the development of interdisciplinary numerical simulation, each participating scientist is a specialist in a certain field of expertise. This makes it necessary to separate the responsibilities for parts of a scientific software-system. For this reason, software engineering techniques have to be applied to achieve a separation of concerns [DR09].

Component Based Software

The term *software component* was proclaimed at the first software engineering conference in 1968 [NR69] initiated by the NATO. The idea behind these components can be traced back to a paper about mass-produced software components, published by M.D. McIlroy [McI68] at that very NATO conference.

Today, component based software engineering (CBSE) is an established paradigm which gets evermore part of standard software development processes [CR06, CH01, EF03, Szy97]. CBSE promise to improve the modelling of complex distributed systems. Interesting goals of CBSE are *separation of concerns*, *reuse of software artifacts* and *reliable conformance with requirements*. These goals also count for the development of scientific applications. There are some component frameworks available for scientific purposes [Nie07a, KKPR01, AAae06, GAL+03] which were used in different scientific software applications [HM06, Mae06, Par06, Lae04, GAL+03].

Code-reuse influences virtually every stage of a software development process starting with the requirement analysis, ending with the runtime of the target system [CH01, CE00, BHB+03]. This review surveys the state-of-the-art in CBSE with scientific computing as domain for application.

Parallel, High-Performance and Grid Computing

Scientific computing is a field for application and development of parallel, high-performance software, since the challenge of solving complex numerical problems is inherent.

Parallel and High-Performance Computing

Complex computational problems have to be solved on powerful computers, which almost naturally provide parallel processing. Amdahl's law for the speedup of parallel algorithms states:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Here the speedup S depends on P, which is the proportion of workload which can be processed in parallel, and N, which is number of parallel execution units. It is obvious that speedup is a very sensitive property[1] and heavily requires optimisation. High-performance computing is the domain for techniques concerning optimal (parallel) processing. This makes parallel programming as well as high-performance computing important for computational sciences.

[1] If $P = 0.5$ the maximum speedup can not exeed 2.

Grid Computing

The term *grid computing* was manifested for the first time in [FKT99] and proclaims an infrastructure for the transparent usage of computer resources, which are distributed in terms of organisation, institution, management and space. The vision of the grid is an infrastructure like the power grid which, in contrast, transparently serves computer resources instead of electric power. The metaphor highlights the importance of transparency since nobody who wants to use an electronic device has to know how — not to mention where — the electric energy is supplied. The user's device only has to have a compatible plug.

This vision of a future computer-infrastructure has some hidden drawbacks, as it is not generally possible to hide communication-time, which plays an important rule in distributed computing. Nevertheless, the vision of the grid attracts many scientists from many fields which expect it to be the future environment for managing computer resources [BKS05]. The served resources are often computational, but can equally well encompass storage, connectivity, data, sensors, actuators [Ce06].

Coupled Simulation

Many interesting areas of active research in many domains of engineering do not consider certain physical phenomena (like fluid- or structure-dynamics) separately, but in interaction. This research is motivated by a large set of possible applications; modern engineering often requires the consideration of coupled systems: optimise a turbine, reduce the fuel consumption of a vehicle, reduce in-vehicle vibration or sound generation, consider the stability of buildings in case of environmental disasters.

To support computer aided engineering in this fields, simulators for different phenomena, e.g. software from computational fluid dynamics (CFD) and software from computational solid dynamics (CSD) have to be coupled. Partitioned analysis therefore aims at reuse of available software for coupled simulation, which implies great challenges in software engineering.

Chapter 2

Software Reuse

The acts of the mind, wherein it exerts its power over its simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another, so as to take a view of them at once, without uniting them into one; by which way it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence; this is called abstraction: and thus all its general ideas are made.

JOHN LOCKE, AN ESSAY CONCERNING HUMAN UNDERSTANDING (1690)

2.1 Relation between Learning And Programming

Many different programming paradigms emerged since machines became programmable. The power of programming languages can be measured regarding the difficulty to express knowledge in terms of the language. Today programming languages are not just tools for implementing algorithms, but they play the important role of providing an interface between human understanding and machine processing. This reveals a challenge in language design: Programming languages have to be simple enough to be processable by a machine and expressive enough to be used by human.

The development of programming languages has started with technical languages, each specialised to a certain machine. It evolved to languages providing mathematical abstractions like functions and types; in this way, routines are coded directly into static procedures, without an interface-layer separating domain-terminology and processing.

Polymorphic-types later introduced support for hierarchical classification of types; related instances can be used transparently; procedures are orchestrated at execution-time (reducing the performance).

Parameterised-types allow to model hierarchy-free type-relations and functions; highly efficient compiler-optimised procedure-composition is provided. This is state of the art of high-performance programming today. The objective of the evolution of programming languages is to increase the expressiveness and efficiency of programming languages: *say more with less words*.

This creates new problems. An expert in a certain domain of expertise can not explain the same knowledge or ideas with the same words to different people. People knowing *his* terminology understand differently than others. An expert *understands more with less words*. New higher programming languages evolve, focussing on domain specific terminology and models.

The influence of this evolution can be seen in many areas of program development. In the beginning of programming the process of inventing a program was perceived as an art, it evolved to be a very technical task and today, gets separated into different problem domains:

- The development of domain-specific languages by software engineers for providing generators for translating model descriptions to executable procedures.
- The use of domain-specific languages by domain experts to specify the procedure in a domain-specific way.

The goal which carries all these developments is the reusability of programs, to understand this is the goal of the first part of this report.

2.1.1 A Definition of Reuse

For understanding the difficulties of program-reuse, this section draws a parallel between reuse in our natural environment and reuse of computer-programs. Reusing existing concepts is an essential part of human creativity.

Reuse as Cognitive Task Examples are ubiquitous in our experience and appear trivial:

- using a stone as knife
- using a knife as screwdriver
- using a chair as ladder
- making fire
- ...

All these points are examples for human creativity and instances of reuse. The ability of humans to observe, to abstract and to learn are ingredients required for these achievements. *Re-use* is not only the application of the same knowledge —in the same way— more than once[1], but cognition is involved. An equivalence between a stone and a knife obviously implies cognition.

Reuse is strongly dependent on a context; if a child has learned how to turn on the light in a room by using a light switch, the reuse of the knowledge in another room requires abstraction from the known situation (or context), to make the general idea reusable in another context. *Abstraction* means therefore to identify and isolate a general idea from a set of specific realisations.

The role of Language The mankind invented another important tool for transferring knowledge: language. If language is involved it is important to signify items using names. Concepts, processes, relations, problems or other entities have to be signified to be describable in language. In natural human language different classes of signifiers exist; nouns to donate concepts or verbs to name actions. These signifiers are used to describe relations in time and space, to describe interactions between entities and which ever; *you name it*.

The role of artificial Languages Artificial languages have to be much simpler. Machines have to be able to interpret them using a finite (closed) set of interpretation rules. Programming languages provide naming of variables, memory, functions, classes, objects: items processors perform on. An important difference between natural languages and programming languages is therefore obvious:

- natural languages are used between cognitive individuals
- programming languages are used by cognitive individuals and interpreted systematically by a machine

Artificial languages are therefore not used to transfer knowledge[2], but to describe processes using formal models.

[1] This is the way in which machines solve problems
[2] Computers (Machines) do not know anything.

Reuse of Computer Programs The goal to make programs reusable is a *holy grail* of computer science and language design and does not come for free. As in our natural environment, reuse of programs is a cognitive task which requires an understanding of the item under consideration; its context requirements, its function and interface must be made *visible*. Reuse by third parties requires additionally a common terminology, consistant models and systematic (reproducible) implementations.

We will see in the next sections, that programming paradigms evolved along the road to more formalisation, context-modelling.

2.2 Reuse in Software Systems

It is caused by intuition, that reuse seems to be a rather trivial task; we reuse concepts in our daily life when we for instance use a knife as a screwdriver. The situation changes completely if we consider reuse of computer programs. The concepts, processes and models they implement are *coded* and relations between them are not explicitly observable. Often algorithms are interwoven with data-structures and sub-algorithms and interfaces between subsystems are not defined, boundaries of sub-systems can not be identified which means that subsystems interfere. Every programmer has a different understanding of the problem under consideration and chooses different names for concepts, relations, processes and objects. Additionally, every programmer has a different idea of how to implement things, which language construct to use for solving a given technical problem. Understanding a program implemented by a foreigner is sometimes comparable to understanding the world view of that person.

Programming models reduce the degree of freedom of programmers and introduce a common proceeding in translating a given model-description into program-code. In the history of programming paradigms, many different programming models were invented. Programming models support software development by prescribing a common proceeding and terminology and therefore a common understanding of what a program is. Programming models use metaphors like *object* or *component* for describing abstract data-types to make programmers think in real world terms.

It is hard to give a closed definition of computer-programs. For decades programs were thought of as implementations of algorithms transforming a given input to an output. A modern view of computer programs has to deal with interprocess communication, human-machine interactivity, artificial intelligence, reactivity, which can not be formulated with a closed algorithmic view on programs [Weg97]. This rising requirements explain a manifold development in programming models and paradigms.

The following section summarises the main path of the evolution of programming paradigms. The focus is thereby put onto paradigms in computational science.

2.2.1 Implications of Reuse in Software Systems

Reuse of software does not have the trivial goal of saving time for reimplementation, which even turns out to be delusive. If assumptions on a program change, its code most likely needs to be reimplemented. Think of a highly optimised program which is written in a non-portable assembler language. The attempt to reuse this code on another platform may enforce the use of an emulator to interpret the code which is far from being optimised. Reimplementation would be the only meaningful action; here not the code, but the algorithm would be reused.

Instead, reusing software and developing software for reuse focuses on:

quality enhancements reused software is tested (by use) [Beh00, Cou98]

amount of maintenance components can be maintained independently [Beh00] even by third parties [Szy97]

separation of concerns especially interesting for scientific software [KKPR01]

parallel developments multiple projects reuse the same component [Beh00]

product development costs components must not be reinvented [Beh00, Gri98, Szy97, Cou98]

commercial components can be sold independently [Szy97].

2.3 The long rocky Road to Software Reuse

Modern software systems implement different functionality; graphical user interfaces (GUI) , program execution environments like operating systems, numerical simulations, office applications or video games. All these software systems have in common, that they are realised with the help of algorithms which are recipes to produce predefined output from given input.

To reuse software is a very fundamental idea. The development of reusable algorithms started with the development of assembler languages, which made machine commands human readable. Later programming paradigms became more abstract from the machines, and compilers had to be used to transform more abstract statements into executable machine programs. This approach made it possible to write programs in a more abstract way, hiding the instruction encoding. These higher programming languages provide predefined structures for general purpose programming tasks; for example loops or parametrised subprograms are provided. The introduced abstractions hide technical details as register allocation, naming and jumps in the program sequence. Abstractions make programming less error prone and the resulting programs better understandable.

The introduction of support for explicit interfaces into several programming languages extended the possibility of making subsystems exchangeable (for a discussion of this aspect see [Cou98]). A superordinate concept is *programming with language independent components*, provided by software component frameworks (see [Szy97]). These black-box paradigms support a functional decomposition of complex systems into independent subsystems. It is applicable if considered subsystems deal with a certain functional task, for which a specific common interface can be defined for a class of subsystems.

An interface for making a software subsystem reusable can only be defined, if it does not need to be changed to provide full functionality in different contexts; the *interface* of a subsystem to its environment has to be constant. To efficiently compose software from black-boxes, a point of minimal effort in composition *energy* must be found[3]. If a program provides functionality which varies in a functional sense, a constant interface can often not be found and a black-box view on the software can therefore not be provided. Reusing such codes in a classical programming language with its focus put on functional system composition is most likely inefficient or not maintainable; the code has to be reimplemented.

This part of the text focusses on the discussion of various programming paradigms and gives an outlook to non-classical programming.

2.3.1 Overview: Approaches to Reuse

The book [Beh00] proposes a classification scheme for approaches to reuse, which is depicted in figure 2.1. The scheme considers four categories of aspects; the *goal of*, the *proceeding in*, the *subject of* and the *generality of approaches* to software reuse. Which are summarised in the following.

Goals of a Reuse Approach Goals of approaches to reuse can be categorised in aims at development *with* software reuse and development *for* software reuse. Approaches to *development with reuse* focus on development of new software with reuse of already available software. In contrast, approaches to *development for reuse* focus on development of software (sub-)systems for reuse in other applications. Both approaches should not be viewed independently but the goals are distinct.

[3] Here *energy* means communication, programming and computational overhead.

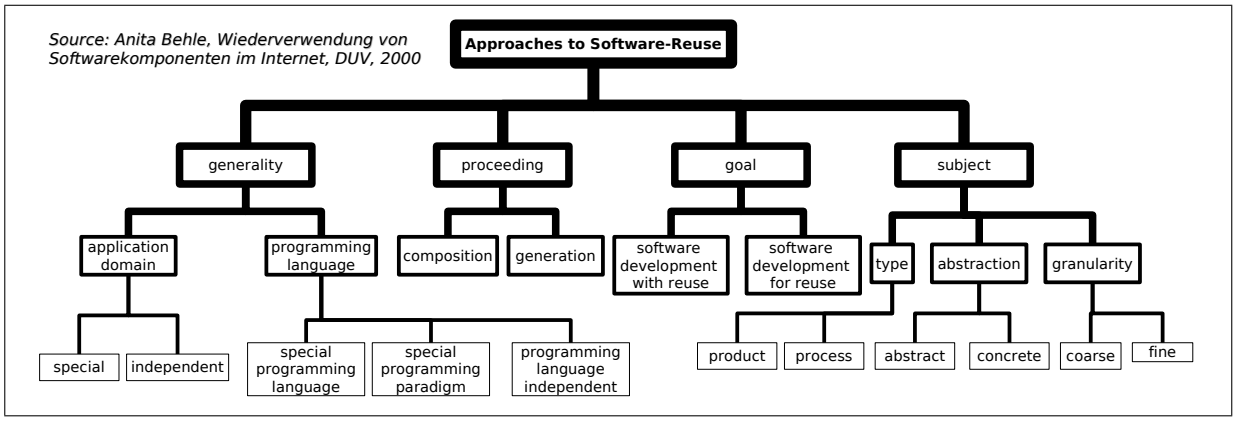

Figure 2.1: *Classification of Approaches to Reuse [Beh00]*

Proceeding in a Reuse Approach There are two different proceedings which can be used in software-development with focus on reuse. One proceeding is based on software components which are independent functional entities being used in composition and the other proceeding (re-)uses generators which generate code from special model descriptions; in this case the model and the code generators are reused. Generative programming is reviewed in chapter 2.3.3 and proceedings of composition are reviewed in chapter 2.3.2.

Subject of a Reuse Approach Approaches to reuse can aim at software products or on its development process and can thereby focus on concrete implementations or abstract patterns. The focused subjects can be fine up to coarse grained.

Generality of a Reuse Approach On one hand approaches to reuse can focus on a special application domain (e.g. [Ber00]) or they can be domain independent (e.g. CORBA, CCA, CTL,...); they can also be programming language independent (e.g. Pattern [GHJV95]) or dependent (e.g. Java Packages [AGH00]) or they can enforce a special programming paradigm (e.g. Component paradigm in chapter 2.3.2.5 on page 18).

2.3.2 Software Paradigms

Reuse is a topic in software engineering since a NATO conference in 1968 [McI68]. At this time M. D. McIlroy formulated the idea of reusing software with a metaphor on integrated circuits which were at this time already subject to reuse. The idea evolved over proximate decades into applicable solutions for the problem of software reuse.

Each new programming paradigm proposed new concepts on how a software system can be modelled and understood. The book [Cou98] gives an overview on the realisations of reuse in different programming languages, but the story of the book ends with object oriented programming. Object oriented languages (e.g. Eiffel[4]) in fact implement the concept of logical separation of subsystems by interfaces. Nowadays approaches to reuse go beyond this aspects supporting distribution of subsystems, providing a high degree of transparency and language independence, platform independence and interoperability [KKPR01, Szy97, AAae06, Gri98].

2.3.2.1 Categories of Classification

Sometimes the question of which programming paradigm is good and which not appears to be more a religious question of belief than a question of judgeable facts. Programming paradigms are abstract constructs and their

[4]In the book [Cou98], Eiffel gets a score of 60 of 60 points for reuse

implementation in programming languages, compilers, interpreters and supporting environments are manifold. From theory of computer science we know that the set of computable functions is covered by each of the presented paradigms, they are in that sense equivalent.

The comparison presented here is based on selected aspects beeing important for programming paradigms in computational science. The most important are *support for parallelisation*, *high-performance* and *maintainability*. Other aspects like *supported granularity for separable artefacts*, the *degree of coupling between subsystems*, *degree of transparency* and *support for optimisation* as well as intrinsic *problems in reuse* are secondary aspects. All these aspects are separately discussed for each paradigm in the following. We consider at first *functional programming*, then *procedural*, *object oriented* and finally *component based* programming.

Some of the aspects like the possibility to parallelisation can not be judged without considering concrete frameworks. At that points the text leaves the abstract level and considers examples.

2.3.2.2 Functional Programming

A function is a mapping of a tuple of input parameters to exactly one unique result. A fundamental concept in functional programming is recursion [AS96]. Functional programming (FP) is a domain for application of so-called divide-and-conquer (D&C) algorithms, which can be easily implemented with the help of recursion.

D&C algorithms exploit that big problems can be often decomposed into smaller ones which can then be independently solved by recursively decomposing the problem again. *Mergesort* is a good example [CLR00], since it demonstrates the potential of recursive algorithms which can be processed in parallel in a very natural way. Reuse within the boundaries of a single functional programming framework is generally not problematic; the paradigm enforces functions to have no side effects. This implies that no global variables can be defined and the context of a function is always completely given through its parameters. All variables of a process are therefor provided by parameters of the function. Reuse of code written in one of the available functional programming languages in another language environments is supported, if the provided framework allows to generate code in a machine near language like C.

It is possible to follow the functional programming paradigm in programming languages which are not purely functional like C or C++; even in object oriented code using so-called function objects or *functors* [Ale01].

Support for Parallelisation Modern functional programming language like Haskell [CLJ+07] implement *pure functions* only. Pure functions are defined to have no side effects and are therefore nice to use for parallelisation, since they are thread-safe[5] by nature. This property of Haskell is used in [CLJ+07] to demonstrate automated code-parallelisation. An assumption of the presented approach is that it requires homogeneous processors to work properly, and the problem has to be reducible in the sense of divide and conquer.

A skeleton based approach to paralellisation presented in [DFH+93]. Each skeleton provides an template for a certain communication and parallelisation pattern.

Performance Theory gives no difference in performance between functional programming and any other programming paradigm. Applications show (e.g. [CLJ+07]) that code written in a procedural programming language is often faster. This may be caused by the fact, that compilers for procedural and object-oriented languages are highly optimised even by vendors of processors. Procedural languages as FORTRAN or C/C++ are more relevant in practise, which makes it attractive for commercial companies to focus on that languages and provide powerful compilers. Approaches like [CLJ+07] show the potential power of functional languages in the context of high-performance computing and modern multi-core-architectures [ABC+06].

[5] A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads.

Granularity Functional Programming is fine grained, it is possible and absolutely reasonable to define a function which only executes one operation (see listing 2.1). Compilers can *inline* the operation to make the explicit definition invisible in further optimisations, but this proceeding is not applicable in recursive functions which can not be inlined.

```
1 inline double add(const double& a, const double& b)
2 {
3   return a+b;
4 }
```

Listing 2.1: Granularity in Functional Programming – C++

Coarse grained systems are a weakness of pure-functional programming. The paradigm enforces functions to have no state which implies that their complete context has to be passed as its arguments. This again implies that the number of arguments to functions tends to be large.

Strength of Coupling Loose coupling of subsystems can be achieved by consequently avoiding definition of stateful functions; this is enforced in many functional programming languages. The avoidance of stateful functions can cause performance issues especially in distributed systems, because old states have to be recommunicated to emulate a stateful function. Modern functional languages like Haskell exclusively implement *pure-functions* and *function forward declarations* which provides the essential techniques for proper decoupling of codes.

The code in listing 2.2 shows a recursive implementation of the function $f(n) = 2^n$ for given n. The MIT-Scheme development environment allows this code to be compiled and reused. In some functional programming languages it is possible to define global variables which may cause problems concerning autonomy of code. This is especially an issue if colliding variable names occur; it also contradicts the *principle of locality* [Rus04] which guards efficient orchestration of subsystems and high performance. The definition of global state variables is more exotic to functional programming and we do not take this into account in the summary.

```
1 ;; This function computes 2^n recursively
2 (define (power2 n)
3       (if (= n 0) 1
4         (* 2 (power2 (- n 1)))
5       )
6 )
```

Listing 2.2: Granularity in Functional Programming – Scheme

Maintainability Modern functional languages like Haskell support software developers in producing well maintainable code by enforcing *pure-functions*. An issue is that coarse grained systems tend to have long function signatures which are hard to handle.

Transparency Functional programming incorporates a stack based processing model which is today implemented on top of the processor architecture of current processors, which have von-Neumann architecture. This abstraction from the hardware implies that technical details of current machine architectures are hidden. This makes functional programming highly transparent.

Support for Optimisation and Customisation The more transparent software development is with respect to the executing machine, the less can be optimised by programmers; developers dependent therefore on good compilers and runtime-environments. Optimisations like *proper tail recursions* in functional languages reduce the natural overhead of recursions [Cli98]. The result is that memory consumption or processor cycles of a program can hardly be influenced by the programmer; high-performance computing is not the focus of any current functional language environment.

Problems in Reuse Most functional programing languages and environments do not support code reuse explicitly.

- most functional languages are not standardised[6]
- side effects / global state may be allowed and used
- functional programming is not intuitive for none specialists
- interoperability with other languages is not addressed

  some languages are only interpreted not compiled

2.3.2.3 Procedural Programming

Procedural or imperative programming incorporates an algorithmic thinking. The computer processor's own languages – machine code – is procedural. The translation of procedural high-level code into machine code is more straight-forward which allows algorithms to be translated into efficient executables.

Procedural programming is often used as a *synonym for imperative programming* (commanding the steps to reach a desired state), but can also refer to the procedural programming paradigm which is based on the concept of procedure calls. *Procedures*, also known as routines, subroutines, methods, or functions simply contain a series of steps to be carried out. Any given procedure might be called at any point in a program, including by other procedures or (often[7]) itself.

In contrast to functional programming, procedural programming emphasise changes in program state and not application of functions [Hug89]. This indicates a more technical view onto algorithms. A procedural program is constructed from consecutive instructions which are executed in given order. The concept of recursion in functional programming is often replaced by loop constructs which often provide a better performance.

Support for Parallelisation Parallelising procedural code is most often done using two different technologies:

OpenMP is only available for shared memory SMP systems [Boa05]

MPI targets both shared and distributed memory architectures [Kal98]

OpenMP is a tool provided by compilers for the languages C,C++ and FORTRAN. It supports programmers in writing thread-parallel programs by providing compiler commands to simplify thread based parallel programming. An OpenMP aware compiler can parallelise canonically coded loops into multiple program-threads. The programmer controlls this automated process with the help of *pragma-annotations* in the source code like demonstrated in listing 2.3.

```
1 #PRAGMA OMP parallel for
2 for(int i=1;i<vectorA.length();i++)
3 {
4         vectorA[i]=scalar*vectorA[i]+vectorB[i];
5 }
```

Listing 2.3: OpenMP parallel for construct

Code with complicated loop structure can not be automatically parallelised and has to be managed manually with help of the OpenMP-API [Boa05]. OpenMP does not address distributed memory architectures.

The Message Passing Interface (MPI) provides another methodology to program in parallel. MPI is based on a procedural view on communication and is most often used for parallelisation of procedural, especially computa-

[6] Though semantics of language constructs are often completely defined in functional languages, syntax varies in many dialects (e.g. Common Lisp, Scheme, Emacs Lisp are all dialects of Lisp with different syntax); other Languages like e.g. Haskell are standardised.
[7] Recursion is not supported in the FORTRAN77 programming language.

tional code. MPI is in a sense very low level parallel programming; programmers control the packaging of data and synchronise parallel processes using explicit synchronisation calls.

MPI has a large community in high performance computing (HPC); users in this field prefer manual handling of communication and synchronisation, because any optimisation trick is applicable.

In the MPI metaphor, communication is modelled as a part of the considered parallel algorithm. This is a big disadvantage of message passing methodology, because algorithmic code has to be interwoven with communication-code. A global view onto the state of a distributed program can therefore not be provided, which causes debugging problems [SGK07]. It is thereby almost impossible to exchange parts of a parallel MPI program without knowing details of it and its implicitly defined protocol.

State of the art MPI implementations address both shared and distributed memory architectures in a transparent way.

Performance Procedural/imperative-programming employ a state-oriented view on algorithms. This view allows a one-to-one translation to machine programs and often results in high-performance. From that reason procedural languages are very popular in high-performance computing and embedded system programming. A disadvantage of this view with a low level of abstraction is a worse code-efficiency in terms of code-metrices [Tha00] which results in high cost of system-development.

Granularity Procedural programming supports a broad spectrum from fine to medium grained systems. It is possible and absolutely reasonable to define a procedure which only executes one arithmetic operation. The compiler can inline the called code to make the definition invisible in further optimisation steps.

As in many paradigms, so-called *modules* can be introduced to increase a separation of subsystems. A software-module is an entity that groups a set of (typically cohesive) subroutines and data structures. Modules can be compiled separately, which makes them in a sense reusable and allows multiple programmers to work on different modules simultaneously. Modules provide modularity and encapsulation, which can make complex programs easier to understand. A separation between interface and implementation is also provided. This makes modules a predecessor of modern component based programming. Components additionally support the distribution of modules.

Strength of Coupling Coupling in procedural code is in general strong; there is no standardised way to make context assumptions explicit. Reusability of procedural code is therefore hard to guarantee. Coupled codes can mutually depend on each others symbols which allows arbitrary strong coupling of subsystems.

Software modules allow to implement technically decoupled subsystems, but there is no guarantee that implicit (hidden) assumptions avoid module exchange [Szy97]. This hidden assumptions are made explicit in analysis and design phases of object oriented software-development processes and are forbidden in any component based approach. Furthermore, module interfaces are only syntactical definitions, they do not reveal semantic of a module i.e. modules can implement an arithmetic subtraction in a procedure named 'add'.

Maintainability Procedural code tends to be unstructured. It is often talked down to be *spaghetti-code* which metaphorically describes that algorithmic code is interwoven with code for data-structures, communication, persistence, and other aspects. A separation of code by introducing explicit interfaces introduces a better separation of concerns [KKPR01], but the application of such approach is not obligatory. The approach in [KKPR01] is optimistic, when the phase of integration of independently developed subsystems is not taken into account and semantics of subsystems are completely ignored.

Transparency Procedure overloading and the definition of identically named functions with different arguments is not supported in the most important procedural languages: C and FORTRAN. Because of missing polymorphic calls, transparency is generally low.

Language extensions like OpenMP provide high transparency with respect to parallelisation, since parallel and sequential code look the same.

Support for Optimisation and Customisation Compilers for C and FORTRAN are highly optimised, but complex applications especially with complex data structures need to be carefully designed to provide good performance [Kal98]. Highly optimised data-structures are often interweaved with algorithmic code which makes them hard to reuse. Algorithmic optimisation on level of modules is only possible by binding interfaces to different module implementation.

Problems in Reuse In procedural programming reuse is based on modularisation. Modules are not defined to be context independent and there is no semantic model to guard semantics of a module, therefore different implementations can not be described in a exchangeable way. The problem of colliding symbol names is also not solved [Cou98].

2.3.2.3.1 Example: FORTRAN A literature review on reusable scientific software would be useless if it would neglect FORTRAN (an acronym derived from The IBM Mathematical Formula Translating System) as the oldest currently used programming language [ASS85]. It is the most popular programming language in the field of computational science especially high performance computing [Kal98]. FORTRAN is a procedural programming language which was originally developed by IBM in the 1950s for scientific and engineering applications. With introduction of the new Standard FORTRAN 2003 object oriented concepts are introduced into the standard, but there are few compilers available, which implement the standard [WG06]. Object oriented concepts advance code encapsulation and therefore reusability of code [Cou98].

There is a big scientific community which uses FORTRAN for software development. There are stable versions of compilers available for each relevant platform and the basic language is stably specified since decades, compilers are powerful and allow their users to implement programs with high performance [Kal98].

FORTRAN provides a simple procedural programming style and does not enforce any methodology to make code maintainable. The focus of the language is on computational application. The productivity of programmers in writing and maintaining complex FORTRAN applications is very problematic. It is bounded to a few lines of code a day [PT97]. To be honest; it is easy to write arbitrarily bad code[8] in any programming language, but FORTRAN almost enforces that. Code written in old FORTRAN derivatives like FORTRAN77 is most likely to be not reusable in other contexts than the original; modern derivatives of the language allow definition of explicit interfaces and other modern techniques to enforce better code, but things like that are not enforced.

Reuse in FORTRAN Reuse in FORTRAN is limited to reuse of libraries which can be seen as a kind of module. The difference between a module and a library is, that all symbols of library are globally defined while symbols in a module are localised [Cou98]. This library based reuse has some issues, since globally defined symbols can easily collide. For example a constant definition $pi = 3.14159265358979$ in a library can easily collide with a less precise $pi = 3.14$ definition of another library, which may cause serious problems and *useless* debugging.

This fundamental issues bring reuse-supporting environments like the Common Component Architecture (CCA) or the Component Template Library (CTL) into play. From the point of view of a FORTRAN developer, these frameworks fix the discussed issues by introducing a new but highly transparent abstraction layer encapsulating

[8] In the sense of code-metrices [Tha00]

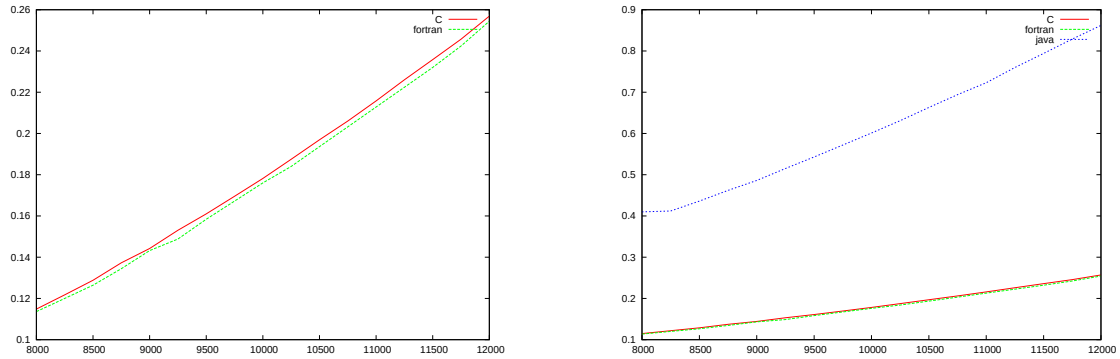

Figure 2.2: *The plot shows the execution-time of differently implemented matrix-vector-multiplications with respect to problem size (matrix dimension). In the FORTRAN (`GNU Fortran 4.3.3`) implementation the operation is implemented directly within the `main`-function — a feature of bad spaghetti-code. In the C (`GNU g++ 4.3.3`) implementation it is implemented using a function. In comparison to JAVA (`Java HotSpot(TM) 64-Bit Server VM, build 1.6.0_12-b04`), both C and FORTRAN provide high-performance.*

```c
void matrix_multiplication(const double* a,
     const double* x, double* b, int n)
{
  for (int i=0;i<n;i++){
    b[i]=0.0f;
    for(int j=0;j<n;j++)
      b[i]+=a[j+i*n]*x[j];
  }
}
```

```
DO i=1,problemsize
  b(i)=0
  DO j=1,problemsize
    b(i)=b(i)+A(j+(i-1)*problemsize)*x(j);
  END DO
END DO
```

Figure 2.3: *Benchmark result from figure 2.2 are made with the above codes. C code for matix-vector multiplication on the left, FORTRAN code on the right.*

given implementations behind an explicit interface. On the other hand does this technology introduce the possibility to use complex (polymorphic) infrastructure services like global arrays [NHK+], network weather service [WSH99], database connections [NKcS03] or resource management [Jü06] in standard FORTRAN code.

FORTRAN Performance One myth[9] concerning FORTRAN is that programs written in this language are faster than others. For this review I decided to consider some examples to get my own experience on this *fact*. Figure 2.2 shows some benchmarking results made with the code listed in listings 2.3. Other results show C faster than FORTRAN, but benchmarking is a hard business and results are sensitive to small changes, so we present only comparison of two identical trivial loops.

2.3.2.4 Object Oriented Programming

Object oriented programming (OOP) allows to implement polymorphic user-defined types. Instances of these types are called objects and can be handled like ordinary data-types (as e.g. `int`, `double`,...). Object oriented programming is derived from the mathematical concept of *abstract data types*[10] [Goo00, Goo01].

A *class* defines the *properties* (also called *fields*) of its *objects*, as well as *methods*. Methods are functions which operate on an objects properties, potentially changing its *state*. The state of an object is represented by the values of its properties. When OOP started to become popular, the challenge of software reuse was thought to be solved

[9] It is ubiquitous in literature on computational science.
[10] or abstract algebra on the set of object-states

[Cou98], since a class can be seen as an interface definition for its instances. The past has shown that this hope was not fulfilled by OOP. In addition to the basic features of OOP, it is important to make contexts explicit [Szy97] and separate interfaces from implementations [Cou98]. The book [Cou98] from 1998 lets the story of reuse end with OOP, without recognising challenges at all.

The Object Management Group (OMG) has been founded to solve problems related to distribution of objects over machines and specified the *Common Object Request Broker Architecture (CORBA)* [TvS03] which provides a distributed object model. CORBA is often stated to be a component model [Gri98], but the OMG describes CORBA as a solution for the problem of distributing objects[11].

Support for Parallelisation An object hides its state information behind the interface of its class definition. Invoking a method on an object is equivalent to sending a message to the object, which makes it intuitively easy to distribute OO systems. For technical realisation of a distributed object system many different solutions are available, most of them are for the popular object oriented programming environment and language JAVA [AGH00]. Java has a bad reputation with respect to performance for computation-intensive tasks [Spw][12].

JAVA provides intrinsic support for implementing thread parallel object-oriented systems [AGH00]. In C++, language extentions like CTL [Nie07a], or the previously discussed solutions for procedural programming have to be used.

JAVA also provides intrinsic support for implementing distributed object-oriented systems (*JAVA-RMI*), but the solution is not interoperable with other programming languages. CORBA and CTL provide frameworks for interoperable distributed systems.

Transparent compiler based automated parallelisation with OpenMP is available for C++ as in FORTRAN and C.

For implementing parallel algorithms, skeleton based approaches are available [KS05], which are comparable to the approach described in [CLJ+07] (has been discussed previously). The survey paper [Kal98] also mentions further examples like pC++[13], POOMA[14], Charm++[15]. Charm++ seems to be currently relevant and there is a large number of further examples of parallel OOP programming environments[16]. Charm++ has been casually picked as an interesting example:

> *A Charm++ program is structurally similar to a C++ program. There are five disjoint categories of objects (classes) in Charm++:*
>
> - *Sequential objects: as in C++*
> - *Chares (concurrent objects)*
> - *Chare Groups (a form of replicated objects)*
> - *Chare Arrays (an indexed collection of chares)*
> - *Messages (communication objects)*
>
> *The user's code is written in C++ and interfaces with the Charm++ system as if it were a library containing base classes, functions, etc. A translator is used to generate the special code needed to handle Charm++ constructs. This translator generates C++ code that needs to be compiled with the user's code.*
> THE CHARM++ PROGRAMMING LANGUAGE MANUAL [PAR]

Granularity Object oriented programming has a broad spectrum of supported granularity. Fine grained systems are in the same way possible as in procedural programming, by using inlining. Coarser granularity is given, when

[11] The CORBA Component Model CCM defines a component model on top of CORBA, adding object life-cycle management and further features.

[12] But results are getting better and [JN04] predicts a break even point of JAVAs performance compared to C performance, though our own benchmark results with a state-of-the-art JAVA environment can not confirm this.

[13] Source on `http://www.extreme.indiana.edu/sage` was last time updated on 09/13/1994.

[14] Source on `http://www.nongnu.org/freepooma` was last time updated on 10-Dec-2004, but seems to be a *stable release* (2.4.1)

[15] active project with successful projects based on it.

[16] [MSM05] lists 218 examples of partly OO environments on page 14

working with class hierarchies. Some environments as JAVA support a coarser level of granularity by providing the concept of explicit interfaces and interface inheritance. Therefore a separation of interfaces from implementations is introduced on code-level [AGH00]. C++ does not provide this feature out of the box; again extentions have to be used (CTL,CORBA,...).

Strength of Coupling A broad spectrum of possibilities for coupling is supported. Reuse of a class in general is not trivial, since implicit assumptions in its code can make reuse impossible. Definitions of pre- and postconditions for each method and method-invocation-protocols reduce these context-dependencies [WK03, ZS06, Som04]. These approaches are already termed to be in the context of software components, OOP and component-based approaches are therefore strongly related.

A class is a autonomous subsystem of a software system, but classes do not need to be modelled independently. A class can be strongly dependent on the behaviour of other classes in a system. That is the reason why class-reuse is often termed white-boxes reuse. Classes in a complex OO system can be defined to be dependent properties of other classes they are related to. This can lead to serious reuse issues making classes strongly dependent [Cou98, Szy97].

Maintainability Code-reuse is intrinsically supported in object oriented programming by the concept of *inheritance*. A dilemma of OOP is, that using these concepts does not mean that the resulting codes are better reusable. If inheritance is used excessively, it makes maintenance hard, since a change in a base class implies changes in its derived classes. If a class hierarchy is badly designed this may cause big problems.

Transparency OO supports intrinsically a transparent way of exchanging type-implementations. A derived class can be used as an implementation of any of its base classes. This achieves a fairly high degree of transparency, but it also encourages programmers to create complex strongly coupled systems, which can hardly be understood [Szy97].

Support for Optimisation and Customisation Generally, optimisation in OO-systems is limited to the inner of a class. Compilers are not able to inline code needing to support polymorphism since this property would vanish otherwise. The support for optimisations inside a class is comparable to possibilities provided by procedural languages. Benchmarks with an object-aware FORTRAN using differently structured code results in ambiguous benchmark results [QRH].

Problems in Reuse Reusable object-oriented code can be realised by so-called *OO-frameworks* focusing on a special domain of application. A code framework contains a class model representation of (all) important entities which are concerned in the model of the given domain. Frameworks are one way to do object oriented reuse [Szy97, Beh00]. The reuse of a framework is limited in the sense that two different frameworks can only be combined if the domain of application of both is orthogonal, meaning that the frameworks do not interfere. The reuse of frameworks is also purely white-box reuse meaning that internal details of a framework need to be known by the user.

2.3.2.5 Component Based

Component based software development is discussed in chapter 3 on page 27. The following section summarises properties of this programming paradigm in short.

Support for Parallelisation Component-based programming can be understood as an extension of OOP: definitions of software components are compatible to the class model of OOP. The most popular approach to parallel component systems is therefore given by JAVA (*thread* on shared memory machines) and JAVA Remote Method

Invocation (Java RMI on distributed memory machines). The limits of this JAVA component interoperability; JAVA RMI as well as the threading model can not be reused in programs not written in JAVA. Highly transparent and interoperable parallelisation is an intrinsic feature of software component technology.

Granularity Generally, components have a broader spectrum in granularity as classes in object oriented programming. Fine grained component approaches are common in component oriented languages as *component pascal* [Hug01] and in GUI development, where buttons, text-boxes and that-like are components [Gri98, WRH02]. Coarse grained components representing complete applications can be found at the other end of the spectrum. Modern distributed operating systems are compatible with many definitions of software components.

Strength of Coupling Each component framework aims at decoupling of codes and provides a non-intrusive way of decomposing appropriately designed software-subsystem.

The autonomy of software components depends on the component framework, which is used. In component frameworks as CTL, components are very autonomous, they can be represented by an operating system process, a thread or can be bound at run-time from a shared object or linked by the linker at compile-time. An operating system process which can be handled in standard ways[17], is the most autonomous entity of computer processing.

Maintainability Since codes are decoupled through interfaces, maintenance can be localised to each separate component. Maintenance of a complex component system with many components can be reduced to exchange or reconfiguration of components.

Transparency Transparency depends on the framework implementation, CTL implements a very transparent component model, where the code to use a component is exactly the same as if a standard OO class would be used[18]. Other frameworks like the CCA, CORBA and any other commonly used instance do not provide this feature, here connections between components need to be established explicitly on a more or less technical level.

Support for Optimisation and Customisation If precompiled components are to be used, compiler based optimisation is obviously not supported. Components may be optimally compiled for a given machine, but they are black-boxes, where shortcuts can not be made between two points in the code to optimise a process.

This does not mean, that component based software can not be optimised, but the parameters for optimisation are different. Components have to be coarse grained to minimise possible communication-overhead. That is the reason why component based software has to be carefully designed. If an component-architecture is optimally designed for a given application-pattern, it can be easily reused. This introduces another level of reuse, namely architectural reuse, which reschedules optimisation from implementation- back to design-phase. This makes component based software development less agile compared to OOP.

Another possible category of optimisations is the scheduling of components onto resources, which has to be optimised with respect to the components and their interaction. This may require dynamic resource scheduling and component-migration.

For achieving high-performance of a component architecture, the quotient *operations per communication* is important, since it identifies loosely coupled subsystems. Such loosely coupled parts of a system have to be identified in design phase. Speed and scalability of the resulting architecture can be judged by the degree of this coupling, since components have to communicate and communication-overhead is the lowest, the looser coupling between two components is.

[17] e.g. by kill, ps, ...
[18] This feature of CTL is restricted to clients in object oriented languages.

Problems in Reuse Reuse is the main goal of component oriented software, but a few open problems and areas of research are:

⇝**Semantics** general semantic modelling

⇝**Contracts** quality of service, service level agreements [Jü06]

⇝**Framework** connecting component frameworks – coupling of inhomogeneous component frameworks [Bü07, Dam05]

⇝**Non-Functional** handling non-functional requirements

⇝**Resource-Management** general load-balancing [Jü05, Jü06]

2.3.2.5.1 Example: Service Oriented Programming Service oriented programming (SOP) in service oriented architectures (SOA) is currently a buzzword in software engineering. SOP is based on the service pattern which is an architectural design pattern classified as a component design. The idea aims at a combination of architectural software design, software components, interoperability (languages and frameworks) and reuse. All software development phases as requirements engineering, problem analysis, software design, implementation, testing and integration, are integrated in a methodology, making it obviously a business compatible model for software development. Text books like [EF03] give an introductory overview over the topic.

Support for parallelisation Services are distributed, but parallel algorithms can hardly be implemented, since communication performance of service oriented systems is bad[19] and the state of an algorithm has to be held in the service-client, making it impossible to achieve good performance. The goal of service oriented systems is interoperable information exchange not high-performance processing.

Performance Service orientation does not aim at performance optimisation.

Granularity The granularity of service oriented architectures is coarse grained. Services are accessed via interoperable protocols with significant overhead; this forbids application in fine to medium grained systems.

Strength of Coupling Coupling of services to their clients is loose. To use a service a *middleware* is required, which is standardised for optimal interoperability within the SOA framework. A service is defined to have only explicit context dependencies. This implies a service to be stateless (though session-mechanisms exist), so that each call to a service gives the same result. This is comparable to functional programming with the difference that services are distributed functional units.

Maintainability Since maintenance is a service-local operation, maintainability is good.

Transparency Services are highly-transparent.

Support for Optimisation and Customisation Support for optimisation or customisation on level of available services is low, service internal optimisations are encapsulated upon a slow standardised interoperable protocol.

Problems in Reuse Service oriented programming focuses on reuse; reuse is one of the most important features of SOP.

[19] see section 7.2.4.1.3 on page 50

2.3.2.6 Software Paradigms – The Big Picture

The present chapter briefly summarises the previously discussed implications of programming paradigms. The considered categories are *support for parallelisation*, *targeted performance*, *granularity*, *degree of coupling*, *maintainability*, *transparency*, *optimisation*, and *reusability*. Table 2.1 summarises the results of the previous chapter and gives an overview. Figure 2.4 is more abstract than the table, but the results of this chapter can be seen as an evidence for the statement of the picture.

Table 2.1: Property-Matrix — Correlating Aspects and Paradigms

	functional	procedural	object	service	component
Support for Parallelisation	good	medium good	good	good	good
Targeted Performance	low medium	high	medium high	low	low high
Granularity	fine medium	fine medium	fine medium	coarse	fine coarse
Coupling	loose strong	strong medium	strong loose	loose	loose
Maintainability	good	bad medium	medium	good	good
Transparency	high medium	low medium	medium	high	high
Optimisation Support	low	high	high	low	low high
Reusability	easy	hard	easy hard	easy	easy

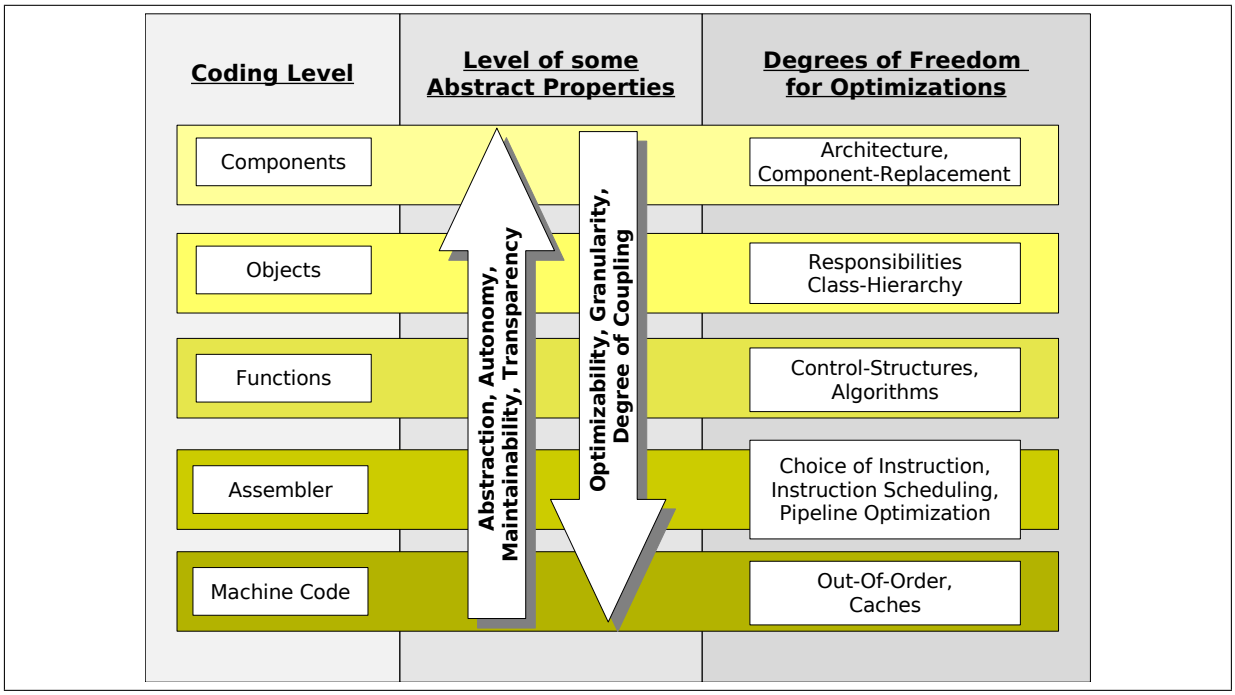

Figure 2.4: *The table shows to what degree a programming paradigm influences aspects like abstraction, granularity, degree of coupling, maintainability, transparency and support for optimisation.*

2.3.3 Generative Programming

Generative programming (GP) is a style of computer programming that utilises automated source code generation through generic classes, prototypes, templates, aspects, and special purpose code generators to improve the productivity of programmers. It is often related to code-reuse topics such as component-oriented programming [JLM00].

In GP a program is not directly implemented in a classical general purpose programming (GPP) language. The focus of GPP is on the implementation of algorithms, i.e. the transformation of input-values to results. In contrast to that, the focus of GP is on *compile-time functionality*, which is in this sense non-functional. Requirements to a generative software system can be implemented at compile-time to optimise performance (loosing flexibility) or at runtime to gain flexibility. Supporting this view of a separation between functional and non-functional requirements, GP is complementary to component based development [Beh00]. In GP, functional units are created with help of a model description processed by *generators*. In classical programming, functional units are implemented directly. We can understand GP as attempt to generalise the process of code compilation. Instead of transforming algorithmic *code* into machine programs[20] the process of translating model descriptions into code is considered.

Figure 2.5 gives an overview of the machine-code-generation processes integrating aspect-oriented programming, generic programming, and classical compilation. This process has three stages which are passed in the given order. These stages are *preprocessing*, *template-code generation* and *compilation*. While the latter one is well known from the last chapters, the others are not yet mentioned. These phases are those, where meta-programming takes place.

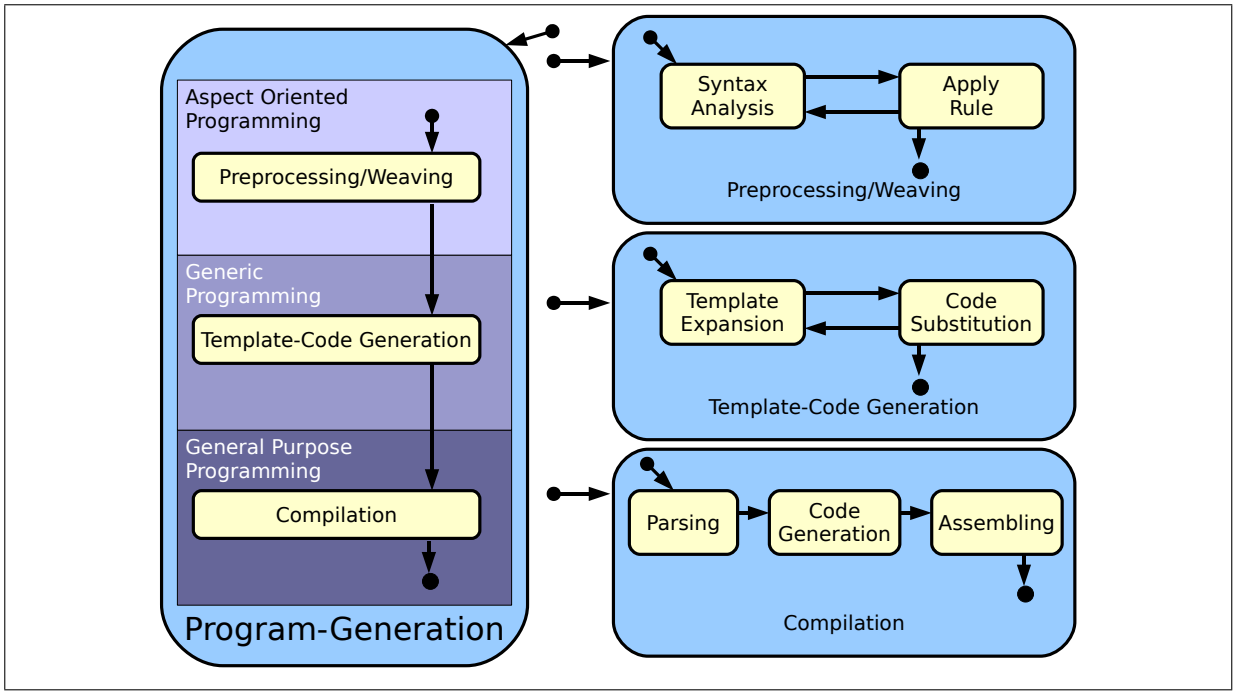

Figure 2.5: *The figure shows a state-chart of a generalised code-generation process. In* preprocessing *(e.g. aspect-weaving) context sensitive-rules (aspects) are used to manipulate code in a recursive process, while in template-code generation non-context-sensitive rules (templates) can be recursively applied. The compilation-state is a simple linear process without loops.*

The artifacts which are developed and reused in GP are code-generators. Generators encapsulate the knowledge of how to interweave given fragments of knowledge (like a model description) and functional entities [CE00]. In

[20] A compiler is an assembler-generator.

contrast to functional components, generators are actively involved in the build process and not present when the program instance is executed [Beh00].

Two different approaches to GP have been applied in real world applications: aspect oriented programming (AOP) and meta programming (MP). AOP is sometimes referred to as a new paradigm, but it depends on a classical programming language since aspects are non-functional entities, which only describe how to interweave functional code into applications, functional units need to be programmed in a classical general purpose language. This is the reason why incarnations of AOP frameworks are yet always connected to a classical programming language (AspectJ, AspectC++, Aspect#, Spring.NET, ... See `http://en.wikipedia.org/wiki/Aspect-oriented_programming`). C++-meta-programming is a technical approach to provide code-generation. The C++ language [Str00] was one of the first programming language which introduced generic programming in terms of templates. Generic programming allows programmers to specify algorithms and data-structures independent of certain data types. In combination with object oriented inheritance and powerful compilers a completely new level of programming was introduced. Meta-programming (MP) allows to specify reusable structural and behavioural patterns [Ale01, Ber00].

2.3.3.1 Generic Programming

For generic programming (GenP) typed languages are extended with support for parameterised types and functions. A type or function with type-parameters provides *slots* for types or constants, in which specific items can be plugged. If all type-parameters are plugged, the generic type is *specialised* to a parameter-free type, which can be used in an ordinary way. The advantage of this proceeding is, that code can be at the same time flexible and inlined by the compiler to guarantee optimal performance thus it can decide which code-instance to take.

Today GenP is implemented in a number of programming languages[21] with different degree of expressiveness and performance. The C++-template-mechanism is fully flexible and powerful and provides — due to its *turing-completeness* at compile time [Vel06] — a *meta*-programming level. The problem of debugging meta-programs is not yet solved in an acceptable way [EAH+07], thus the expressiveness of meta-languages is reduced in some languages. JAVA provides only a subset of generic programming: *meta-programs* in JAVA always terminate thus the mechanism is not turing-complete.

Some general-purpose MP-languages are embedded into programming languages and others defined in conjunction with a language (AspectJ,...). Specialised meta-programming languages are called *Domain-Specific Language* (DSL) or *executable Domain-Specific Language* [SMH03] and will be discussed in section 2.3.3.4.

The first approach which is presented in the following is *aspect oriented programming* (AOP). The second one is C++-template MP which is discussed in section 2.3.3.3.

2.3.3.2 Aspect Oriented Programming

Aspect oriented programming (AOP) is a programming paradigm with its focus put on separation of concerns. AOP aims at supporting software development by implementing mechanisms to isolate *cross-cutting concerns*, and to handle them separately from functional code. AOP allows to define a new kind of module: non-functional modules termed *aspects*.

An aspect does not yet have a precise definition, this is in heavy contrast to concepts of functional units which are — after decades of continuous development — well defined. The definition of aspects is rather weak but it is still applicable. The idea is based on the design principle that a functional unit or component of a software system should exactly do one thing [Som04] and not more. To fulfil this requirement is generally not easy since code does a lot of things which are not directly concerned with its core functionality:

- logging

[21] e.g. JAVA, C++, Haskell,...

- synchronisation with interacting processes
- communication with other peers
- memory allocation
- ...

All these examples have some things in common:

- they are not core-tasks
- they are non-functional
- they are ensnared
- they can not be efficiently separated into functional units with classical techniques

The concept of aspects solves exactly this problem. It aims at enclosing non-functional concerns into units of composition; so-called aspects, which describe how code has to be interweaved from given code-components. An aspect is therefore a manipulation-rule, which specifies how given code fragments are to be manipulated in order to fulfil some given requirement. Aspect oriented programming (AOP) is very interesting especially for making high-performance software systems maintainable and efficient at the same time [KLM+97].

In one of the first articles on AOP [KLM+97] the given example results in a reduction in lines of code (LOC) by 98% compared to hand-optimised code[22]. These results motivated scientists to adapt the approach, which is now used to combine component oriented with aspect oriented approaches [DRR+06]. AOP promises to be helpful in making MPI-based codes easier to maintain [DRR+06, SS06] and is already proposed to be used in multi-physic simulation to improve flexibility and code-composition [EDE03][23].

2.3.3.3 Template Meta-Programming

Template meta-programming is a programming paradigm in which generic types or functions (termed *templates*) are used for code-generation. The output of template meta-programs are *compile-time constants*, *data structures*, or *complete functions*. The paradigm is supported in a number of languages: C++, D, Eiffel, Haskell, ML and XL.

C++ is picked here as example for a meta programming language caused by its relevance in computational science [Ber00, CE00]. It is of particular interest in high-performance areas, such as image recognition and processing, numerical computing, graphics, signal processing, or libraries with lot of variability and dependencies [CE00]. The language C++ allows to encapsulate code-generators into template-libraries. A good introduction to C++ template techniques gives [Ale01], while [CE00] gives a good idea of models to understand the concept behind templates.

The challenge in GP is to provide clean and intentional interfaces, variability as well as high run-time performance; these aspects are a matter of program design rather than an inherent property of MP. In generative programming variability can be expressed in terms of features. Features of a class may vary at:

weave-time (AOP) e.g. adding logging statements

compile-time e.g. specialisation of a parametrised class

link-time by linking different library implementations

run-time with two common flavours

late-binding by using a late-binding component framework

OO-polymorphism by using polymorphic object

[22] The performance of the hand-coded code stays 4 times faster, but an intuitive implementation was factor 100 slower than the aspect based code [KLM+97].

[23] related to the approach in [Jü06]

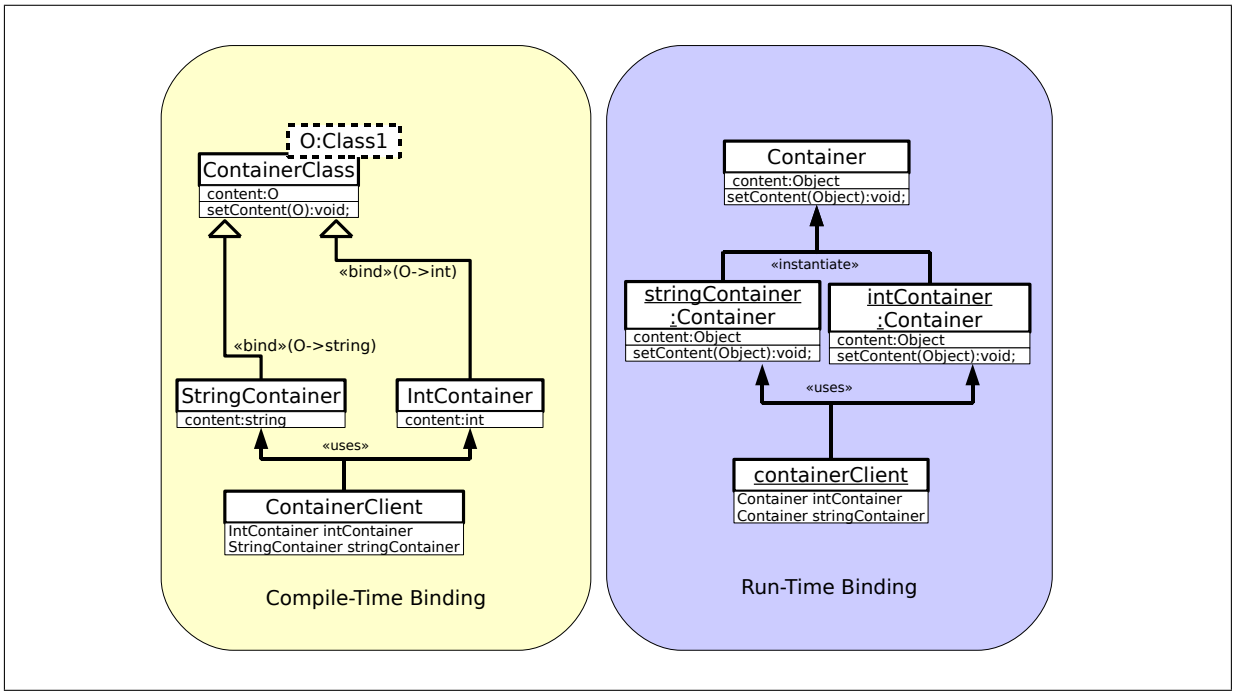

Figure 2.6: *The picture shows two different implementations of a container class. The left implementation uses a parametrised class to define a container and the right one implements an OOP container. The OOP container allows to store any given type as content, which is derived from the general type* Object*. This requires (slow) cast operations and can also result in runtime-problems due to wrong types. The template-based implementation is in contrast completely type-safe, expensive cast-operations are not required.*

The goal of templates is variability of types at compile-time. The template-system of C++ introduces an algebra of types. The set on which this algebra operates is the set of classes, while the operations are given as:

parametrisation changing a certain type to be a parameter

specialisation/instantiation assigning all parameters of a template

partial specialisation substituting a subset of all parameters and thereby introducing a hierarchy of subtypes

An interesting meta-programming concept, which was presented in [Ale01] is the *policy*. A policy encapsulates non-functional code and is therefore the implementation of aspects in template-MP. Though the language for templates is different from the language for aspects the mightiness of both approaches is the same, but both have very own properties.

2.3.3.4 Executable Domain Specific Languages

The last important meta-programming paradigm considered here is *language oriented programming* and focuses on the invention of a specialised language for a given domain of expertise. Such languages are called *domain specific languages* (DSL) [vDKV00]. Some DSLs like the Backus-Naur-Form (BNF) for syntax specification or the Unified Modelling Language (UML) were not meant to be programming languages or executable by a machine, at the time they were invented. Nowadays UML as well as the BNF can be used to generate executable programs. The translation of UML into an executable is yet limited to special purpose applications; often models of business processes. BNF is today an executable language for modelling grammars; language interpreters can be generated from BNF expressions (See lex,yacc,boost::spirit,...). Executable DSLs can be used to provide an interface to the code-generation process, thus they are relevanted in this context. Executable DSLs can be used for generating code for later compilation steps. Interpreters for DSLs can be implemented in GP meta-programming environments

[SMH03], which make DSLs the most important applications of meta-programming techniques. The book [CE00] shows, how analysis, design and implementation phases can be managed in meta-program development processes.

2.3.4 Software Reuse – Summary

The last sections surveyed important approaches to software-reuse. The most important results are the following:

- software needs to be designed for reuse
- developers need to
  - be disciplined
  - follow conventions
- code-documentation is most important (semantics)
- interfaces (compile-time, build-process, run-time,...) need to be specified
- reusable components need to be tested (semantics)

Chapter 3

Components in Software Development

3.1 Motivation – Component Based Software

Software component technology is the foundation for functional unit decomposition. This domain of software engineering is important in computational science because it allows to couple codes written in different, even incompatible, environments. An example is the coupling of a FORTRAN simulation with a C++ optimisation framework.

Current frameworks for software component systems have the following properties:

- support extensibility of complex distributed systems
- allow the coupling of subsystems on incompatible hardware
- encapsulate and protect intellectual property
- allow resource-sharing in complex grid-environments
- allow enhanced support for system-configuration at runtime

Component Based Software Engineering (CBSE) assumes that parts of a software system exist in advance or are to be designed in a reusable way. This correlates with the setting of software development *with* or *for* reuse, respectively. System development processes in CBSE focus on integrating available components rather than developing them from scratch [Som04]. This has some implications:

- technology to bind a client to a component implementation is required (component framework)
- independent components have to be available (developed)
- techniques to describe the topology (architecture) of a composed system need to be present
  - through a *driver* component which is comparable to the main() method in traditional programming
  - through architecture description language (ADL)
  - through control scripts
  - hierarchical declaration (CTL, CQL [Jü06])
- rules for component deployment have to be defined
  - meta-data description for describing components (CQL, Eclipse)
  - standardised component file format (jar (Java Archive File), ctl.so, ctl.exe)
- components have to be distributed on available resources (resource management [Jü06])

- distributed components need to be discovered and found for use

  - the physical topology of the network must fit to the system architecture

3.1.1 Theoretical Concepts of Component Based Software

Component based software engineering is the domain of developing software with aim on reuse. Most publications in this domain claim non-formal proposals to model problems of reuse, though more formal approaches are available for decades [WOZ91]. There are some theoretical results, which are important for understanding what a reusable software component is. This results are summarised in the following.

3.1.2 Interfacing

A fundamental part of a reusable system-specification is the definition of a system-interface. All component frameworks provide a DSL for this purpose, the Interface Definition Languages (IDL). These IDLs can be used to define the structural interface of a component specification describing in full detail the syntax of all methods the component provides, including their parameters, result types and possible exceptions. Referring to the book [LG86], which covers the algebraic specification of abstract data-types, [Som04] presents the elements of a formal specification.

1. The functional unit (system) has to be signified by a unique *name*
2. All exported operations of the system are to be defined in a *signature part*
3. A set of axioms that characterise behaviour of the system is to be defined in a *axioms part*

   defining the semantics of each operation

   relate the operations used to construct or manipulate entities of the system

   a state protocol [ZS06]

The modelling of semantics by axioms represent an uncertainty; protocols like discussed in [ZS06] can be used to define the set of legal sequences of method-calls between a client and a data-type. These protocols can be represented as finite state machines in a client-server setting, or as context-free grammar in peer-to-peer, callback or message-bus settings. These formalisms have not yet found practical application in software development, instead of giving this precise definition [Som04] introduces a rule of thumb for the number of axioms which are to be defined.

3.1.3 Terminology

The description of software, which is based on or built with components has its very own terminology. There is no problem to find a dozen definitions for any entity which is related to the component based design of software, but it is impossible to find one, which everybody would accept as complete. To be honest, you can hardly find pairwise consistent definitions of what a software component is.

3.1.3.1 Definition of Software Component

Here are some examples of definitions for software components:

> *A software component is a specific, named collection of features that can be described by an IDL component definition or a corresponding structure in an Interface Repository.*
> SEE OMGS CCM SPECIFICATION [OMG06]

> *A software component is a composable element of a component framework.*
> SEE MARKUS LUMPE ET AL. IN "TOWARDS A FORMAL COMPOSITION LANGUAGE"[LSNA97]

> *A software component is an independent unit of software deployment. It satisfies a set of behaviour rules and implements standard component interfaces that allow it to be composed with other components. These behaviour rules are often specified as design patterns that-must be followed when writing the component.*
> SEE ROB ARMSTRONG ET AL. IN "TOWARD A COMMON COMPONENT ARCHITECTURE FOR HIGH-PERFORMANCE SCIENTIFIC COMPUTING" [AGG+99]

> *A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*
> SEE BILL COUNCILL AND GEORGE T. HEINEMAN IN "DEFINITION OF A SOFTWARE COMPONENT AND ITS ELEMENTS"[CH01]

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*
> SEE CLEMENS SZYPERSKI IN "COMPONENT SOFTWARE - BEYOND OBJECT-ORIENTED PROGRAMMING"[SZY97]

Especially the last given definition of *software component* (from Szyperski [Szy97]) is broadly excepted in literature. It is common sense in literature that the statement "*explicit context dependencies only*" in this definition also states that a component has no externally observable state (see e.g. [Som04, Sch99]). However this is generally unacceptable for computational components since a great loss of performance would result. We tend not to interpret the statelessness to components. For the domain of scientific computing the following statement seems to be satisfactory.

> *Software components are building blocks from which different software systems can be composed.*
> SEE KRZYSZTOF CZARNECKI AND ULRICH W. EISENECKER IN "GENERATIVE PROGRAMMING: METHODS, TOOLS, AND APPLICATIONS" [CE00].

Krzysztof Czarnecki and Ulrich W. Eisenecker who gave this definition justify the simpleness of the definition by explaining that it is not only futile, but harmful to come up with a general classical definition for software components.

> *We have a classical definition for objects. A classical definition means that we can provide a single set of necessary and sufficient properties to define an object: An object has an identity, state and behaviour. The concept of a "component" has a completely different quality. It is an example of a natural concept, rather than an artificial, constructed concept. According to the theory of concepts (see Appendix A in [CE00]) we can easily construct concepts using a classical definition — "objects" and many mathematical concepts are constructed, classically defined concepts. However, most natural concepts do not have a classical definition, that is, you cannot find a single set of necessary and sufficient properties defining them. As an example, try to define the concept of a "table". Is a "table" a top with four legs?*
> SEE [CE00].

Not only every author redefines the term but also every framework defines its own mechanisms for composition bordering the use of a software component to the domain of a framework [Bü07, Dam05]. This stays to be a big issue of component based software. The next explanation wants to give a better idea of what a component is:

> *These formal component definitions are rather abstract and do not really give you a clear picture of what a component does. One of the most useful ways to consider a component is a standalone service provider. When a system needs some service, it calls on a component to provide that service without caring about where that component is executing or the programming language used to develop the component.*
> SEE [SOM04].

3.1.3.1.1 Characteristics of Reusable Components The Book [Som04] further gives two critical characteristics of a reusable component:

1. *The component is an independent executable entity. Source code is not available, so the component does not have to be compiled before it is used with other system components.*
2. *The services offered by a component are made available through an interface, and allow interactions are through that interface. The component interface is expressed in terms of parametrised operations and its internal state is never exposed.*

SEE [SOM04].

For the context of scientific software the first characteristic given in the last citation does not hold since scientific code is often available as source code only. The focus in almost all definitions and characterisations of software components is on independent deployment e.g. to make software components a commercial interesting good [Szy97]. This is not the primary goal of software in computational science. For scientific components it is important to:

- achieve a separation of concerns
- provide simple tools for the construction and control of complex software systems
- support software tuning to achieve high performance in performance-relevant subsystems (kernels)

  - provide the possibility to introduce independent kernels for optimisation

  - allow a loss-less coupling of highly optimised subsystems

  - provide the possibility to manipulate the component bindings for achieving the best available performance
- provide the possibility to define architectural patterns which can be easily reused by untrained programmers allowing them to use highly optimised software

  define a set of standard interfaces for describing architectural patterns

  implement a set of important core components
- support FORTRAN, since it is still an important programming language in the engineering domain

Observation—Making Available Software a Set of Components Making an existing scientific software a black-box software component is not a trivial task since the standard way to build scientific software is not compatible to this paradigm, because:

- the state-of-the-art in CBSE does not provide a standardised proceeding to describe context

  context dependencies are not defined explicitly and are only known (if at all) personally by the developer

  semantics of provided functionality is not specified explicitly
- scientific software often provides a framework which has to be *specialised* and encapsulated behind an interface to become a component, this results in less flexible software
- reuse has to be planned [Beh00] and is an explicit feature of a software

The meaning of the term *composition* varies broadly, you can establish the composition of two software subsystems at compile-time, link-time, or run-time and there are further subdivisions of these categories. The term component based software development and all available frameworks aim at run-time composition. A component framework provides possibility to compose independently written codes at run-time into an application.

By source-level components [Jon02] a fourth category of binding-time is introduced. Composition at build-time is often not considered in publications on CBSE, though it is a very important point in open-source software development, and thereby in computational science.

3.1.3.2 Component Model

The definition of *component model* is given implicitly in most publications, by referring to an example. Another issue is that this term is sometimes misleadingly defined in collision to the term *software architecture*. This is unacceptable for the aim to define a common terminology.
In conceptual terms, a component model is a definition of how different parts interact with one another within one granular space. Translating the big picture definition into Java APIs is a more difficult task. A component model becomes both an overall architectural plan as well as a set of individual APIs that enable programmers to realise the vision. [Sri97]

This is a bad example for a good definition but there are better definitions in literature:
A component model is a definition of standards for component implementation, documentation and deployment. [Som04]

Component model implementations provide:

- Platform services that allow components written according to the model to communicate;
- Horizontal services that are application-independent services used by different components.

To use services provided by a model, components are deployed in a container, which provides a set of interfaces to access the service implementations of a component [Som04]. This definitions captures the ideas of component models and gives a good basis for discussing some examples for available component models in 3.2.1.

3.1.3.3 Component Framework

At first some definitions of component frameworks, found in literature:

A component integration framework is an implementation of a set of interfaces and rules of interaction that govern the communication among components.[AGG+99]

A component framework is a collection of software components with a software architecture that determines the interfaces that components may have and rules governing their composition [SL96]

A component framework is a software entity that supports components conforming to certain standards and allows instances of these components to be 'plugged' into the component framework. The component framework establishes environmental conditions for the component instances and regulates the interaction between component instances. [Szy97]

Component frameworks are the tool-box which allows to decouple software from its context on the technical level. Each component model has its very own framework implementations which implements the model and therefore provide both, mechanisms for coupling compatible components and horizontal services.

3.1.3.4 Component Architecture

A unique definition of *component architecture* is again impossible to find. The most interesting thing is, that even mix-ups can be found in the literature. The introduction of the Common Component Architecture [AGG+99] gives the following definition:

A component architecture is a specification of a set of interfaces and rules of interaction that govern the communication among components and other necessary tools, such as repositories and composition tools.[AGG+99]

The book [Gri98] gives a longer explanation of what a component architecture should provide. This description is fairly long but it will be summarised at this point: A component architecture should support components in achieving the following properties:

- *Independence of the environment*: programming language, operating system, network technology, development environment
- *location transparency*: local or remote usability invisible for the user
- *separation of interface and implementation*
- *self-describing interfaces*: for the coupling at run-time, a component should give information about its possibilities and access-points (syntax and semantics)
- *plug and play* a component should be instantly usable after installation
- *integration and composition* a component should be composable to a bigger system

This description is far away from an axiomatic definition; the statements are to weak and in some sense contradictory or impossible. To get a distributed plug and play environment a component architecture must know a lot about the network architecture, available components, semantics and many more; there is no component architecture available which fulfils these requirements.

Both of the given definitions are – compared to broad literature – more the definition of a component model, than the definition of an architecture. The confusion of the CCA-Team may be a result of the fact that the design of the CCA is abutted to the Common Object Request Broker Architecture (CORBA) which really defines a standard architecture for the *Object Request Broker*-Pattern. In the rest of the literature a component architecture is not defined in this sense, whereas the term *software architecture* is defined as e.g.:

> *Overall design of a system. An architecture integrates separate but interfering issues of a system, such as provisions for independent evolution and openness combined with overall reliability and performance requirements. An architecture defines guidelines that, together, help to achieve the overall targets without having to invent* ad hoc *compromises during system composition. An architecture provides guidelines for safe system evolution. However, an architecture itself must be carefully evolved to avoid deterioration as the system itself evolves and requirements change. The right architectures and properly managed architecture evolution are probably the most important and challenging aspects of component software engineering.*
>
> SEE PAGE 544 IN SZYPERSKI [SZY97].

This definition motivates the following definition of a *Component Architecture*: A component architecture is a software architecture as defined in [Szy97]; it additionally defines components and defines the binding structure between them. In other words: A component architecture is a tuple $(\mathbb{k}, \Re)$ where $\mathbb{k}$ is a set of components and $\Re \subseteq \mathbb{k} \times \mathbb{k}$ is the coupling relation between the components.

3.2 Example Implementations

This section brings light into the empty definitions from the previous section by giving examples of implementations of the defined terms. Especially interesting are very successful examples of component architectures and models like Eclipse[1].

3.2.1 Component Model

The list of component models is long. Every *big player* in the world of software defined its very own component model. Microsoft defines the *Component Object Model* (COM)[Loo01, TvS03, Gri98]. The Object Manage-

[1] see `http://www.eclipse.org/`

ment Group defining the CORBA Component Model (CCM)[OMG06] on basis of their distributed object model, the Common Object Request Broker Architecture (CORBA)[TvS03, Gri98]. *Enterprise JavaBeans*[Pie99] is a specialised component model for the modelling of distributed business processes, it encapsulates for example database entries. *JavaBeans* is a general component model which supports only JAVA as implementation language, not making it interoperable. Voyager is a modern powerful component model having the same drawback as JavaBeans, it is not language independent[Pie99, Gri98]. An other component model is the Common Component Architecture[AGG+99, AAae06], which defines an high performance aware component environment. The Component Template Library (CTL)[Nie07a, Nie07b] is a C++ template library[2] which implements an easy-to-use and flexible component framework.

3.2.2 Component Framework

Each component model listed above is implemented in form of a component framework. An exception is CORBA and the CCM which are specifications of component models implemented in a big number of frameworks like OpenCCM[3], Qedo[4] and many more.

3.2.3 Component Architecture

The best and most successful example of a component architecture is *Eclipse* — an extensible IDE and Rich Client Platform [CR06, GB03]. Eclipse is based on a component model and framework called *OSGi*, which aims at late extensibility allowing to define or to implement so-called *extension points*.

3.3 Basic Component Techniques

CBSE is based on a few basic concepts and techniques which are discussed in this section. One feature of every component framework is the possibility to compose components into an application at run-time. Additionally most frameworks provide comfortable mechanisms to compose complex applications from available components. Some approaches do focus on an overall architectural description. For doing so the framework needs to find adequate components for a given purpose. In distributed systems this yields another important aspect which has to be considered — the architecture of the parallel computer on which the component system is executed. The mapping of components onto resources is not a trivial task and can generally not be automated, and completely dynamic load-balancing may carry all system optimisations *ad absurdum*.

3.3.1 Late Binding

Late binding means that codes can be dynamically orchestrated at run-time. Well known approaches which support exchange of machine code at run-time are shared objects [5] and dynamic-link libraries (DLL)[6]. The problem with this mechanisms is, that they:

- depend on the operating system

  are therefore not standardised in a portable way
- are for machine-local usage only
- no standardised selection mechanisms

[2] like e.g. boost or loki [Ale01]
[3] `http://openccm.objectweb.org/`
[4] `http://qedo.berlios.de/`
[5] in UNIX
[6] in Microsoft Windows

- provide a flat namespace for libraries
- versioning is not managed (see DDL-hell[7])

In a component approach late-binding mechanisms are extended to provide:

- language independent polymorphism

  - components can be used from clients written in a different language and a different paradigm

- remote access to components
- provide the possibility to discover, select and exchange components at run-time

  - component discovery can be used to introduce e.g. a version management, ...

- since the component framework encapsulates the operating system mechanisms a OS independent view can be provided
- interfaces can be inherited implementing a hierarchical name-space

3.3.1.1 System Composition and Composition Languages

CBSE focuses on the decomposition of subsystems and therefore implicitly requires mechanisms for system composition. Here two different approaches can be found in scientific component frameworks:

hierarchical composition is based on local composition rules and inherits the assumption, that a component using another specific component knows best what properties the component has to have

architectural composition is based on a global view and architectural composition description. It implies that information about available components is present at the design-phase of the component system

The hierarchical composition is implemented in the Component Query Language in CTL [Jü06]. The latter architectural composition is discussed in the literature under the term *architecture description languages* (ADL). ADLs are a very own field of research [Cle96] and will not be further discussed here.

3.3.2 Resource Discovery

In distributed systems resource discovery yields another important aspect which has to be considered in the context of distributed component systems [Jü06]. The architecture of a parallel computer on which a parallel component software is to be executed has to be considered, when the mapping of the components onto the hardware is performed. To ignore the software architecture in the mapping of the machine and in the development of the application is harmful. The mapping of components onto resources is not a trivial task and can not be done automatically [FPSF06]. Resource Discovery is an area in many different contexts:

grid computing [And04, ACK+02, KBM02, AAF+01, CFK+98, FKL+99, CFFK01]

pervasive computing [ZMN05, DWJ+96, KNS+04]

mobile computing [Sat96, DWJ+96]

3.3.2.1 Non-Functional Contracts

Non-functional contracts [Jü06] also called *service level agreements* are a solution for the resource discovery problem. They encapsulate the aspect of coupling software components with respect to their defined properties.

[7] See `http://en.wikipedia.org/wiki/DLL_hell`. The problem is eased in open-source operating systems. Here maintainers do the version control, but the general problem stays.

This concept also gets applied in parallel scientific applications in the context of aspect oriented programming in scientific computing [EDE03].

Combining Late Binding and Resource Discovery Late-binding and resource discovery are strongly related. If late binding is supported, the mapping of code onto resources has to be considered, therefore a resource-discovery is needed [Jü05].

Chapter 4

Motivation for Parallelisation

Reasons for executing programs in parallel are multifaceted. We motivate a closer look to techniques and background by presenting important goals of parallel computer architectures.

4.1 Memory Driven Parallelisation

In memory intensive applications, memory of several machines needs to be unified to create computers with arbitrary amount of memory. This proceeding can also be used to accelerate access to memory, because each processor has local memory with dedicated access channel. Memory driven parallelisation can therefore be used to accelerate algorithms working on large data sets.

4.2 Quality Driven Parallelisation

Business critical applications necessitate a guaranteed quality of service. Quality of service concerns hardware down-time, failures or resource management with respect to quality, i.e. response time, performance or failure tolerance. In this field of application, multiple resources are used to provide *redundancy*.

4.3 Performance Driven Parallelisation

In computational sciences, parallel programs are usually used to reduce the response time of computations or, to increase the size of problems which can be solved. To write parallel programs is often disproportionately more expensive than to write sequential programs; to provide efficient parallel programs is a complex engineering task. It requires detailed analysis of algorithms and often even completely different algorithms, which are harder to understand and maintain than sequential ones. The potential increase in performance provided by parallel programs is fragile; the performance is sensitive to induced overhead (for example through communication cost) and often not each part of a program can be efficiently parallelised at all. For a parallelisation to have any significant impact on the overall execution time of a program, parts which can be scheduled in parallel need to be much larger than sequential ones.

4.4 Hardware Driven Parallelisation

The maximum number of operations processors can execute is proportionally to its clock rate. As long as processor industry is able to increase clock rate of their processors, computers and programs they execute get automatically faster. This development is naturally limited by laws of physics[1]. These physical limitations put an upper bound on the clock rate of processors which is proportional to the speed of information propagation inside the circuit.

In history of micro-processors, the main mainspring of accelerations was the reduction of structural scales in the circuits, which continuously reduced the distances which information had to travel. This distance has physical limitations, because structure cannot be shrinked infinitely. An example in the next section will illustrate that.

4.4.1 Sequential Super Computer Reference

Currently the lower bound of the size of an average half-pitch of a memory cell in industrially producible processors is at 45nm ($45 * 10^{-9}$m). Light travels this distance in $15 * 10^{-17}$s which results in a theoretical upper bound of $6.666 * 10^{15} = 6.666$ Peta Herz for the clock rate which is more than a factor of 10^6 above current technological limits. We will now play a small intellectual game by considering technical data of a virtual 6.6PHz computer:

Current processors already consume more than 100W of power; our virtual PHz reference would consume something in the magnitude[2] of $100 * 10^6 = 100$ Mega Watt. A modern nuclear plant would be enough to supply for 12 of such virtual super computers. The processor of such machine would produce so much heat that a cooling (if possible at all) would be hard.

A problem which was not yet considered in our virtual scenario is that the core processor is not at all the bottleneck in modern computer architecture. The real problem is memory access in sequential machines; this will be discussed later.

4.4.2 Parallel Super Computer Reference

The previous discussion shows that scaling clock rate is physicall limited. An alternative of increasing clock rates is to increase the number of processors. This also results in more possible operations per time. This approach—in contrast—has the advantage that the complete architecture including memory is scaled; bottlenecks as described above can be avoided. The foundation for parallel computers are networks which allow parallel processes to communicate; this technology is available. State of the art parallel super computers have over 200000 processors and have ability to process $1026 * 10^{12}$ floating-point operations per second. The computational power is just a factor of $\frac{2}{13}$ away from our sequential reference, but the power consumption including memory, harddisks, and network is at 2.3455 Mega Watt; the machine has a much better efficiency than ours and is already in existence[3].

4.4.3 From High-End to Desktop

The previous description shows potentials of parallel processors. Microprocessor industry has reached a point in the development of processors where an increase in clock rates is not efficiently possible anymore. Current desktop processors already have four sequential *cores* working in parallel. In near future most likely more cores will be preferred; increase in clock rates will stop being the driving force.

Parallelisation therefore becomes an obligation to exploit power of future computers; hardware driven parallelisation is thereby induced as new challenge into software engineering.

[1] The speed of information has an upper bound i.e. the speed of light.
[2] The power consumption of processors is approximated by $P = CV2f$ where C and V depend on the technology. The power consumption is obviously proportional to the clock rate f.
[3] Data has been taken from TOP500 List - June 2008

4.5 Summary

Each of the given goals of parallelisation has its own solutions and problems. Next sections will summarise parallel computing in contexts of high-performance computing and grid-computing. Both fields aim at very different goals; high-performance computing aims at parallelisation for performance reasons, while grid-computing aims at homogenisation of inhomogeneous, globally distributed resources.

Chapter 5

High Performance Computing

High performance computing (HPC) is a discipline in computer science which focuses on the optimal usage of state-of-the-art and future super-computers. In history of HPC, the leading computer architectures changed continuously[DSSS05, ABC+06]. In the first TOP500 [1] from June 1993 the most popular system architecture were symmetric multiprocessors (SMP), the second most were massive parallel processors (MPP), followed by single processor and SIMD (single instruction multiple data) machines. This setting evolved to a broad basis of clusters followed by MPP and constellations nowadays. Almost 75%(72% in 11/2006) of the systems listed in one of the last TOP500 lists[2] are clusters, 21% of the systems are massively parallel and 4%(6% in 11/2006) are constellations. Listed clusters have an average of 1344 processors while the biggest one has 65,536 processors with a local 2GB memory each [Tea02]. The network topology of that machine is a blend of a local torus combined with global tree structure.

Current publications are dealing with problems concerning scalability of chip-technology in terms of clock frequencies. The scalability of a single processor core has achieved saturation, making it necessary to scale future computer architectures in terms of number of processors [ABC+06] instead. Moores law predicted in 1965 that speed of the fastest computers at a time doubles every year or two [Mol06]. Past has shown that this self-fulfilling prophecy is a guideline for the semi-conductor industry. Translating the law into multi-core scalability predicts an upper bound of 1024 processors available in computers of next decade. Current parallel programming techniques do not allow to program such systems or to automatically distribute computational load onto thousands of cores. This forces the HPC-community to focus on new programming techniques [ABC+06] for the *many-core problem*.

5.1 Parallel Computing Architectures

In the beginning of computer architecture, *John von Neumann* invented the machine model depicted in figure 5.1 [Neu00]. This concept is nowadays the foundation for almost every computer architecture. John von Neumann introduced a central control unit, a central memory unit and a compute unit. This so-called *load-store architecture* does not need to be completely sequential. The von-Neumann architecture has evolved into so-called pipelined super-scalar architectures dividing large sequential logic system blocks into smaller ones allowing the clock rate to increase and more than one operation to be performed at a time. Pipelined processor architectures are intrinsically parallel, to reduce the physical size of individual subsystems. This reduction in size reduces the time information needs to travel through the logical unit and therefore results in higher possible clock rates.

Super-scalar architectures introduce parallelism not only for increasing clock rates. The idea of a super-scalar architecture is to introduce more than one execution unit into the system architecture; the control unit can schedule

[1] See `http://www.top500.org`
[2] As of 06/2007

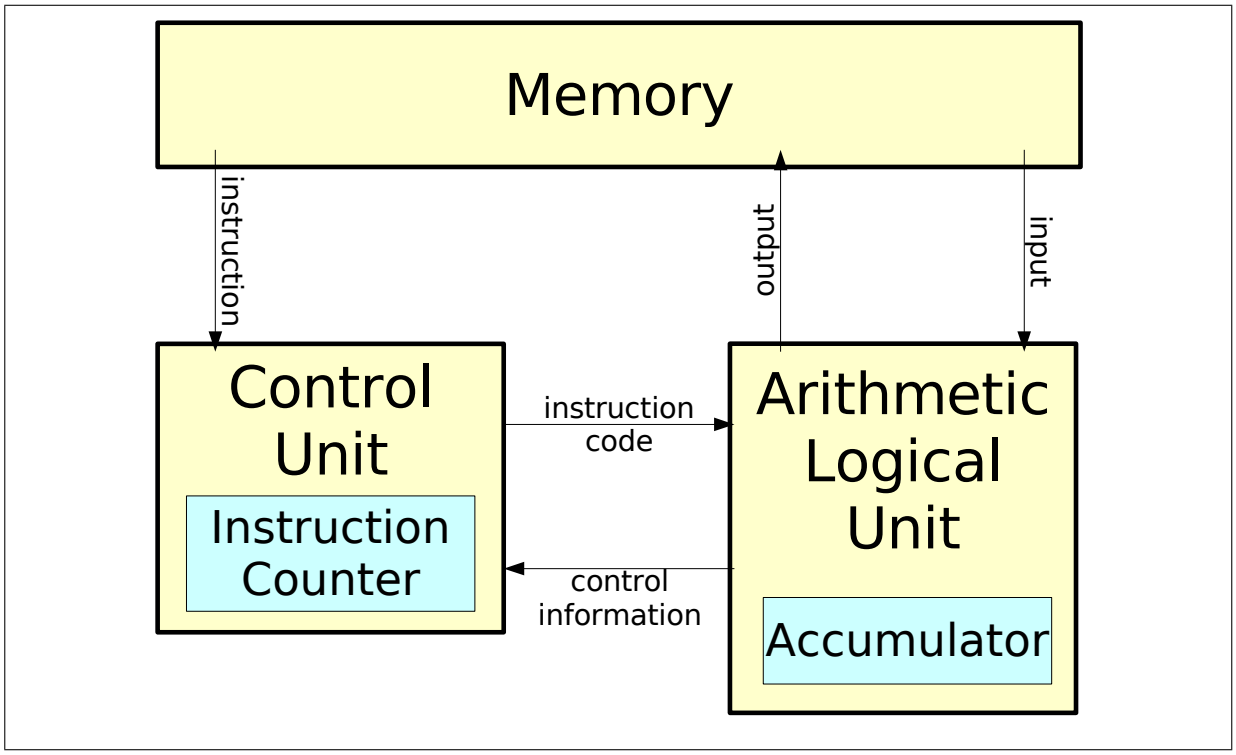

Figure 5.1: *The picture shows the concept of the sequential von-Neumann architecture. The arithmetic logical unit (ALU) is controlled by the control unit which reads instructions from memory, decodes it for the ALU and controls execution. The so-called von-Neumann bottleneck is the interface to the memory since every instruction implies a number of loads and stores from memory.*

the execution of operations onto multiple resources. Every modern processor architecture is super-scalar and pipelined.

5.1.1 A Simple Performance Measure

The von-Neumann bottleneck is the reason why memory intensive computations are slow; a memory access operation is much slower than arithmetic operations: An already old-fashioned Power5 Processor has a performance of more than 15 GigaFLOPS which means that it can do $15 * 10^9$ operations per second. That means that one operation lasts $6.667 * 10^{-11}$ seconds. At the same time does a memory access in most current memory systems cost more than $20 * 10^{-9}$ seconds and up to $100 * 10^{-9}$ seconds in benchmark settings [GJWJ07]. Ignoring a cache hierarchy which can mitigate this problem for successive (local) memory access patterns, in the time for one memory access current machines can execute between 300 and 1500 arithmetic operations[3].

A modern standard 3 Ghz processor has around 8 floating point units, which results in a theoretical peak-performance of $3 * 10^9 * 8 = 24$GigaFLOPS, but this values are never achieved because of the described memory bottleneck.

In distributed computing the overhead of communication is even an order of magnitude higher (around factor 1000). This allows at least around 1.5MFLOP in the time of the communication of one single scalar variable. This relations can be translated into the following performance measure, since performance of a program obviously correlates to the measure 5.1 which is equivalent to the measure 5.2 on shared memory machines.

[3]Theoretical speed of current processors is even much higher than the used $15 * 10^9$ operations per second.

$$\text{Performance is proportional to } \left(\frac{\text{Operations}}{\text{Communications}}\right) \tag{5.1}$$

$$\text{Performance is proportional to } \left(\frac{\text{Operations}}{\text{MemoryAccesses}}\right) \tag{5.2}$$

5.2 Granularity

Granularity is a measure of the size of components that make up a system. Systems of large components are called coarse-grained, and systems of small components are called fine-grained. As one can see from given examples of parallel architectures, it is possible to find and exploit parallelism of the finest grain to implement transparent instruction level parallelism. This kind of parallelism has limited applications and not every application gains performance. To run programs optimally, their loop structure has to be analysed and modified. Memory access optimisation plays an important rule; good compilers can help a lot but they cannot automatically find the global optimum of possible performance. Nevertheless, today they have evolved to powerful tools but leading scientists predict a paradigm change towards a new level of granularity [ABC+06].

5.3 Instruction Level Parallel Machines and Flynn's Taxonomy

There are some orthogonal architectures for computers with more than one processing unit. By far the most common way to classify architectures the taxonomy provided in [Fly72]. Flynn classifies computer architectures according to the number of instruction streams and data streams they work on. The first categorisation is on instruction level. Parallel architectures with a single instruction stream controlling a *parallel processing unit* are called *single instruction* (SI) controlled. Another possibility are parallel architectures, in which the PPUs are all controlled by a individual instruction stream. These architectures are called to be *multiple instruction* (MI) controlled.

Orthogonal to these concepts of control are the concepts of data. A computer architecture may allow its processing units to work on single data (SD), while other architectures allow them to work on *multiple data* (MD). By combining these concepts one gets four different architectures; *single instruction on multiple data* (SIMD) and a *multiple instructions on multiple data* (MIMD) architectures are the most common. Single instruction on single data architectures are non-parallel (von-Neumann [Neu00]) machines.

5.3.1 Single Instruction Multiple Data

Modern processor architectures have SIMD processing units (IBM: AltiVec, Intel: SSE, AMD: 3Dnow!) which are used for instruction-level parallel computations. Today, SIMD devices are getting even more popular. General purpose 3D-graphic accelerators and physics accelerators developed for games evolve into interesting new platforms for applications with fine-grained parallelism [NVI07c, Cle07b].

Benchmarks in matrix-matrix operations give impressive results, but the performance on general less-local operations like matrix-vector are disappointing. Nevertheless these architectures will gain importance in supercomputing [NVI07b, NVI07a, Cle07a].

5.3.2 Multiple Instruction Multiple Data

MIMD architectures have evolved from fine grained special purpose architectures [4] into coarse grained thread-level parallel architectures explained in the next paragraphs. Alternatives to these architectures are so-called *very long instruction word* (VLIW) computers. These architectures allow the compiler to code very long instructions, which means that there is enough information in the instruction available to make the processor aware of parallelism. VLIW architectures can be seen as the successor of MIMD architecture with reduced control complexity. The modern successors of the VLIW architecture are *Explicitly Parallel Instruction Computing* (EPIC) architectures (e.g. Intel Itanium architecture[HMR+00]) which provide VLIW with restriction to in-order execution.

5.3.3 Vector Computers

Another important computer architecture for parallel computations are vector machines. Vector machines are strongly related to pipelined super-scalar architectures. The special goal of vector machines compared to those architectures is their focus on the memory bottleneck. Vector machines mitigate the problem by having a massively parallel memory architecture, making machines very expensive and power consuming. The big goal of vector machines is that their memory access is parallelised [Sch03]; the memory bottleneck is therefore dilated.

5.4 Shared Memory Parallel Multi-Processor Machines

All previous given architectures extend instruction level parallelism. A more coarse grained parallelism can be found in multi-threaded (Hyper-threading in the Intel Pentium 4) or multi-core architectures (IBM Power, Intel XEON, SUN Niagara, AMD Opteron). These architectures implement a thread level parallelism which results in less process switch overhead (multi-threading) or even completely parallel processing of thread execution (multi-core). Every thread has its own associated instruction stream. Threads generally share the memory allocated in their process context. This has the unwanted side-effect of making race conditions, cache coherency problems and thereby asynchronous fail behaviour possible.

The possibility to work with multiple threads on a common shared memory founds the multiple threads single/multiple data (MTSD/MTMD) paradigms. The roadmaps of all general purpose semiconductor vendors has changed towards those architectures [ABC+06] since their old strategy to satisfy *Moores Law* [Mol06] by scaling in terms of clock cycles has reached a technological status of saturation and inefficiency. Future general purpose processors will allow a large number of parallel threads to be executed [ABC+06]. To satisfy Moores Law in the near future one can expect state-of-the-art processors of next decade to have up to a thousand of parallel general purpose processors providing a large number of parallel executable threads. All such architectures with more than one processor and a shared common memory are also referred to as symmetric multi-processor (SMP) machines.

5.5 Distributed Memory Architectures

All the mentioned architectures are too fine grained if the problem scales in terms of large grained parallelism. That is why state-of-the-art super-computers have a cluster architecture. A cluster consists of a number of individual computers (nodes) with an individual operating system running. These nodes are connected by a high-speed network [Slo05]. For a large number of processors these networks do not connect every individual cluster node with every other[5]. Instead of using a fully connected network, there are alternatives with other, not that expensive network topologies[6]. Commonly used topologies for large scaling clusters are trees especially Fat-Trees, Toruses,

[4] See Cray T3/E

[5] Full wiring would have n^2 interconnection point which is impracticable and too expensive in practise.

[6] Topology in this context is a binary, symmetric "*is-directly-connected*" relation on the set of cluster-nodes.

Hyper-Cubes and hybrids of them. For smaller clusters other topologies are in common use: binary trees, arrays, rings, stars or even fully connected networks.

5.5.1 Virtual Single System Image

A strategy to make the use of a cluster manageable and transparent is the *single system image* (SSI) [BCJ] approach. SSI provides a software layer which combines the cluster nodes to a virtual single machine with a single virtual shared memory. The management and migration of processes is done by a distributed operating system layer, which encapsulates all management [BL98]. The parallelism in this machine model is comparable to the SMP approach, but the access to local memory is much cheaper than an access to remote memory, which makes it impossible to have a thread parallel execution. The granularity of threads is too fine for such *none uniform memory architecture* (NUMA) machines. Parallelism is on process level, meaning that inter-process communication (IPC) has to be used explicitly through IPC mechanisms like sockets, Unix pipes or MPI [TvS03, Cal96]. Efficiency of SSI architectures is, because of the automatisms in scheduling, strongly dependent on the application. The future development of OpenMOSIX, which is the most important MOSIX implementation for Linux clusters, has been cancelled on 1st March 2008, because of new technologies like multi-core and virtualisation making the solution superfluous.

5.5.2 Multiple Programs Multiple Data Cluster Programming

Coarser grained non-transparent approaches to cluster programming work with single/multiple programs working on single/multiple data. The SPMD or MPMD approaches gain their parallelism on program or process level. A SPMD/MPMD program consists of many parallel running processes which communicate over a network to exchange data. This approach allows to work on multiple or single data. This flexibility is a goal of abstraction, since hardware is not programmed on instruction level any more, but on program level. The communication in the HPC context is in most cases done over a special application programming interface for message passing, which is called message passing interface (MPI). Many MPI implementation like the Local Area Multi-computer (LAM)[7], MPICH[8] or OpenMPI [9] are available. OpenMPI is a new component based MPI framework, meaning that all layers of the framework are implemented in exchangeable components [SL03, WGC+04, GFB+04]. There are also implementations available which are highly optimised on underlying hardware; these are provided directly from hardware vendors. MPI targets the medium to course grained parallelism where every communication (send/receive) is stated explicitly in the program code. It represents thereby a procedural style parallel programming which also founds the usage of MPI in high-performance computing, where procedural languages especially FORTRAN are extremely popular.

5.5.3 Future Trends

It is quite obvious that the idea of clusters, namely to couple independent but connected machines to a coupled machine, is a software problem. The evolution from strongly coupled parallel super-computers like the *Thinking Machines CM series*[10] or the *Cray T3E*[11] to more loose, software coupled clusters is obvious[12]. Although the newest machines again consist of special purpose cluster nodes[13] the overall architecture as remains to be a cluster. The trend seems to point in the direction of hardware-abstractions, virtualisation techniques and software-based coupling. That is why newer approaches to HPC are targeting into direction of simpler programming and simpler communications.

[7] `http://www.lam-mpi.org`
[8] `http://www-unix.mcs.anl.gov/mpi/mpich`
[9] `http://www.open-mpi.org`
[10] in the 1980s
[11] In 1995
[12] 75% of 2007 top 500 super-computers are clusters
[13] E.g. in HP BladeCenters or the IBM BlueGene architectures

The very explicit style of programming in MPI programs makes it hard to reuse parallel programs in different contexts. There is no standard way to couple two individual parallel MPI programs, instead the code of both components has to be changed and the program must be merged to a new monolithic implementation combining the computations of both parts. That is the reason why newer approaches target into the direction of decoupling to gain the reuse of high-performance parallel software components. These component based approaches are founded on a so-called multiple/single-component multiple/single-data model (SCMD/SCSD). They have the potential for high-performance parallel applications combined with an easier to model (and therefore easier to understand and maintain) communication behaviour.

Extrapolating this tendencies into the future, a component based view on software, which is the next logical step towards more abstract architectures, appears to be very promising.

Chapter 6

Peer-to-Peer Computing

Today one of the fastest supercomputer with about $1,000,000$ [And04] continuously working nodes is the SETI@home project [ACK+02]. SETI@home is a project which uses idle computer resources of registered members for data analysis. The processed analysis tasks are completely independent from each other, making it possible to be run massively parallel. The project started in 1997 with most users being connected to the internet over a slow 56k modem. The working packages have 350KB[1] of size; enough data to keep a typical computer busy for about a day but small enough to download over even slow connections in a few minutes[ACK+02].

Today the SETI@home project is based on a framework for public-resource computing named BOINC [And04]. The relationship between projects and participants in these projects is asymmetric, meaning that the the project can use the participants resources but not vice verse. Malicious behaviour such as intentional falsification of results must be handled by the system. Using untrusted resources founds the need for redundancy in all computations. Machines can loose connections can be powered off, and many more things can go wrong. All such failure situations are handled by the BOINC framework, thus providing an interesting solution for large sets of independent tasks.

[1] About 1 minute over an exclusively used 56k connection.

Chapter 7

Grid Computing

The term *grid computing* was manifested for the first time in [FKT99] and proclaims an infrastructure for the transparent usage of computer resources, which are distributed in terms of organisation, institution, management and space. The vision of the grid is an infrastructure like the power grid which, in contrast, transparently serves computer resources instead of electric power. The metaphor highlights the importance of transparency, because nobody who wants to use an electronic device has to know how — not to mention where — the electric energy is supplied. The user's device only has to have a compatible plug.

The provided metaphor has some hidden drawbacks, as it is impossible to hide communication-time in a computer network, which plays no rule in the power grid. Nevertheless, the vision of the grid attracts many scientists from many fields which expect it to be the future environment for managing computer resources [BKS05]. These resources are often computational, but can equally well represent storage, connectivity, data, sensors, actuators [Ce06]. The intrinsic differences between grids and other distributed environments is that grids must support user and resource abstraction [NS02]. Many computer scientists are interested in grid-computing from a scientific point of view, thinking about new visions, challenges and solutions [BFH03, NSW03].

Currently, there are a number of competing *grid frameworks* in use. One is the *GLOBUS Tool Kit* (GTK) defining an *Open Grid Service Infrastructure* and *Architecture* (OGSI/OGSA) [Fos05]. Another is the *Legion* project supported by the University of Virginia, which is a meta-system software project using object-oriented technology, providing a single, coherent virtual machine that shall address issues of scalability, programming ease, fault tolerance, security, and site autonomy [GHN04]. Another available grid framework is *UNICORE*, which overcomes the common difficulties by providing a uniform interface for job preparation and control which gives in a sense seamless and secure access to supercomputer resources [Rom99]. Scientific computing is one primary application field for grid computing, since the challenge of solving complex numerical problems is inherent in this field. Complex computational problems were always solved in parallel on super-computers, therefore methodologies for applying grid-like techniques are already available, tested and adaptable [HPR05].

7.1 Use Cases for Grid Technology

Use-cases for a grid infrastructure include computational resource sharing, virtualisation and globalisation of storage and file-systems. An overview of a number of visions related to grid computing can be found in [BFH03]. Every involved scientist has its own vision of the grid, but there are a few general categories in which research takes place; the computational grid for combining computational power, the general resource grid for giving access to central resources like sensors and actuators and the data grid providing a framework for miscellaneous storage problems. These three categories are also the three basic issues in a computer architecture. The computational grid is equivalent to a processing unit, the sensors and actuators are equivalent to input/output-devices and the data grid is equivalent to storage. Any possible grid solution is therefore a resource sharing solution providing

distribution transparency across organisational boundaries. Since the grid is in this sense a globally distributed operating system merging heterogeneous resources into a global representative – namely the grid — every other existing solution for related problems evolved into grid solutions (e.g. CFD-Framework [AAF+01], distributed operating system [GBL04], component framework [GKC+an], batch system Condor-G [TTL02]).

The grid community is currently in a consolidation phase [BKS05], ideas have been developed and need to be applied to user friendly, stable frameworks for special purposes. One of those consolidation projects is the European EGEE initiative[1] in its *glite* middle-ware. Glite aims to be a usable grid implementation, but with its current state lacks portability, since it is only available for an already obsolete Linux distribution. Other frameworks like the open grid service infrastructure (OGSI) based on the open grid service architecture (OGSA), which is the basis of GLOBUS, have—like the European UNICORE project—a more conceptual nature.

7.2 Basic Concepts and Technologies

A complex distributed infrastructure like the computational grid needs a lot of technology to provide a transparent, self-determined, user friendly and secure environment. Contributors have to be able to provide and restrict access to their own resources. The concept of *virtual organisations* in a grid allows this [MSZ06]. A user or institution can be a member of a virtual organisation and user/group mechanisms like in all modern operating-systems are provided. Authentication and authorisation are essential concepts in every multi-user operating-system. The difference between the user-management in an ordinary operating system and in virtual organisations is the distributed character making it essential to use cryptographic methods.

7.2.1 Virtual Organisations

A virtual organisation is the basic entity for managing authentication and authorisation in a grid infrastructure. One virtual organisation can have several members and resources. With their help, the authentication for services or resources can be delegated to other authorised virtual organisations.

7.2.2 Public Key Infrastructure

Public-key-cryptography considers asymmetric cryptographic methods. Asymmetric means in this context, that two different keys are used for encryption and decryption. If the key-space is big enough, it is quasi impossible to decrypt a message if the decryption-key is not available. For using a public-key method an asymmetric pair of keys must be generated, one private key and one public key. The private key is the secret which a user has to protect. The public key can be freely distributed and used for encrypting massages which are addressed to the user who knows the private key. The private key owner is the only one who is able to decrypt the message [Wät04] again. These asymmetric methods can be used not only for implementing a secure channel between two peers, but also for authentication. If Alice[2] wants to proof that she is Alice, she can ask Bob to give her something she can encrypt with her private key. If Bob can decrypt the message again with her public key, he can be sure to communicate with Alice. By having a public key of another person one can write him confidentially and authenticate him as the right person. The following paragraphs basically is a summary of [BP01].

7.2.2.1 RSA and Public Key Exchange

Basic problems are solved since the publication of the RSA algorithm in 1977. Problems concerning key exchange and an infrastructure for distributing and verifying public keys are called *Public-Key Infrastructure*s (PKI). The

[1] See http://public.eu-egee.org/

[2] Alice and Bob are the most famous cryptographic persons. Alice is the long form of the letter A, and Bob is the long form of the letter B. If A sends a message to B a cryptographer would always write Alice sends a message to Bob.

most persuasive implementation of a PKI is based on the X.509-Standard for public key certificates. .

7.2.2.2 Hash Functions

Another important cryptographic technique are hash functions, which are surjective functions to map data to a so-called hash value. A hash function should be (strongly) none-colliding; two different input values should quasi always have a different hash value. The target set of a hash function generally has fewer elements than the source set, this means that collisions are generally possible, but the propability for collisions should be at minimum for strongly collision free hash functions.

Hash functions can be used to verify if data has been changed between two checks function evaluations.

7.2.2.3 Signature Algorithm

A digital signature algorithm is a combination of a hash function and a public key algorithm. If Alice wants to sign a text she generates a hash value of the text and then encrypts this value with her *private* key. This encrypted hash value is Alices' signature for the text. If the text is changed the hash value of the text will change and the signature will lose its value. If Alice sends the text with the signature to Bob, then Bob can verify her signature by decrypting it, generating the hash value again and comparing the two values. Therefore it is important for Bob to know the hash function which Alice used for her signature.

7.2.2.4 X.509 Certificates

A X.509 certificate contains a number of additional information compared to plain public keys. In the current version (v3) of the X.509 standard it contains:

Algorithm identifier Identification of the used signature algorithm.

issuer Information to identify a certification authority. (CA)

validity Information about the time of validity of the certificate.

subject Information to identify distinguished the certified end user.

Subject Public Key Info Identification of the public key algorithm and the users public key.

Issuer Unique Identifier (Optional) The use of this field is not recommended (RFC2459).

Subject Unique Identifier (Optional) Used in conjunction with X.500 directory service.

Extensions (Optional) Used for extensions.

Certificate Signature Contains the signature of the CA.

A X.509 certificate allows therefore an authentic public key exchange. A PKI needs a certificate directory where users can query and download certificates. In most open implementations the X.500 directory service is used for this job. The directory can be accessed by lightweight directory access protocol (LDAP).

7.2.2.5 Revocation of Valid Certificates

For implementing the possibility to revoke a valid certificate, a PKI can implement a certificate revocation list (CRL). Newer implementations of the CRL implement an on-line certificate status protocol (OCSP). This provides a service to verify the validity of a certificate just in time of transaction.

7.2.2.6 Trust Models

Trust models describe the relation between end user, trusting parties and CA. Most commonly used models are *certificate hierarchies* and *cross certificate*; both models can also be mixed.

7.2.2.6.1 Certificate Hierarchies Certificate hierarchies are build, when a CA delegates its rights downwards to a hierarchically lower institution. In a company every department may have its own CA and a CA hierarchy may be equivalent to the companies internal structure. The advantage of this model is that a department does not need to trust any other department automatically, but every instance has to trust the highest CA in the hierarchy which is therefore also called the root-CA.

7.2.2.6.2 Cross Certificate So called *cross certificates* are the solution to make (bi-)direct links of trust between two independent CAs. This technique can also be used in combination with the hierarchical model.

7.2.3 Trusted Authority and Rights Delegation – the GLOBUS Security Infrastructure

The GLOBUS security infrastructure (GSI) is a good example for a grid security infrastructure. It aims at interoperability of heterogeneous sites and ease of use. The article [KTB05] gives an overview over the model, points out problems, and presents the GSI solution. The GSI is based on the previously described X.509 standard and implements right delegation, dynamic Virtual Organisation conglomeration and single-sign-on for the user. These three aspects, together with the inhomogeneity of the VOs infrastructure, form the basic problem of grid interoperability and VO formation. The interoperability problem for inhomogeneous sites using different grid frameworks can only be overcome by providing a common standard and derived interoperability gateways. Open standards as the X.509 are the best choice for an interoperability framework. The standard is also used in already established security frameworks like Kerberos [SNS88, J.K93].

With the existing definitions of grid framework, following types of operations are assumed to be supported by GLOBUS security model [KTB05]:

- Allocation of a resource by a user
- Allocation of a resource by a process as proxy for the user
- Communication between processes in different trust domains

7.2.3.1 Single Sign On and Right Delegation to Processes – User Proxy

A user is always a member of a specific domain, or VO. On machines belonging to the users domain the user can log-in and generate a proxy process which represents his identity through holding the users credential. The generated proxy can then use the users credential for utilising dynamically allocated resources and to create other processes [KTB05]. For implementing a proxy timeout GLOBUS generates a new credential for every proxy which is only valid before the given timeout. This temporary credential is signed by the users valid credential (users' private key) and is therefore also valid.

A resource proxy on the server site can evaluate the given user proxy credential and can decide if access to the given resource is to be granted.

7.2.4 Grid Framework

A grid framework like GLOBUS does not only deal with security issues and solutions, though security is an important aspect of the grid. By presenting an overview over the security architecture of the most common grid-framework *GLOBUS*, which is the basis for most other grid projects a central aspect of grid computing is already

discussed. A security architecture or framework builds the foundation of every grid framework. All other services can be build upon the security framework. At this point we will go on discussing the OGSA in more detail.

7.2.4.1 Open Grid Service Architectures

Interoperability does not only depend on security and trust but has a lot of technical issues. The first GLOBUS implementations were based on standard internet socket programming. Sockets guarantee interoperability only on the three lowest layers of the protocol stack, meaning that they allow to establish a reliable logical inter-process communication (IPC) channel, but solutions on higher layers are out of the scope of the mechanisms.

7.2.4.1.1 Web Services and the Pre-WS GTK Encoding of complex data-types and all related problems like e.g. encoding scheme encoding (XML-Schema, Document Type Definition (DTD)) can be done with the Extensible Markup Language (XML), which is an open standard of the world wide web consortium (W3C) [EF03]. State of the art middleware which aims at interoperability in inhomogeneous environments takes this into account and uses XML as basic technology. The W3C also defines a standard for a XML based middleware which is named *Web Service* (WS) standard [EF03]. The GLOBUS implementation changed over the time from the now called *pre-Web Services* or *pre-WS* (Versions 1 and 2) to a new Web Services based implementation of the framework (Version 3). The *pre-WS* variant of the GLOBUS tool kit (GTK) is now considered as obsolete. GLOBUS is only one of a few grid implementations, but WS does lead to interoperable software [EF03], which is primary goal of the grid.

7.2.4.1.2 SOA, Scientific Computing and the Grid The architecture which results if WS infrastructure is used, is broadly called Service Oriented Architecture (SOA). A SOA provides loosely coupled services. These services are often implemented using WS. Therefore SOA and WS are strongly related in the literature. SOA is currently often presented as the holy grale of application integration; this makes SOA called a ”management buzzword“. SOA does not guarantee any success in software development (see Bernd Oestereichs chapter 39 in [ST07] “Mythos und Wahrheit von SOA”). SOA is a hype-word which stands for nothing more than an interoperability framework for stateless software components, where the statelessness guarantees that any two clients do not interfere using a component. In this sense SOA is also strongly related to component based software development which also targets loosely coupling.

7.2.4.1.3 SOA as General Solution for Middleware? The question is: *“Why are SOA and a WS framework not the general solution for future computational environments? Why inventing the wheel again and again?”*. The answer to this question is rather simple; SOA guarantees interoperability. This goal is achieved by the cost of coding overhead; XML and the WS-protocol use a tag based gossipy ASCII encoding standard, and the WS-communication protocol is based on the Internet Protocol (IP) especially on the Transmission Control Protocol (TCP) , which is not a high performance protocol. The result is that SOA is a good choice for interoperability but not for scientific components. Some enthusiastic component framework developers already ignored these facts and implemented an instance of the Common Component Architecture which is based on OGSA and therefore WS [Sria, KG04, Srib][3]

7.2.5 Grid Services – The Backbone of a Grid Implementation

As we have seen a grid service is not a scientific software component. The grid implements the idea of a “globally distributed operating system” and a grid service is therefore comparable to a system call in classical operating

[3] As a result a simple code example (a Centigrade to Fahrenheit converter) with only one line of logic-code needed 241 overhead lines of code (LOC) resulting in a horrible code-efficiency of $\frac{1}{241} = 0.41\%$. A intuitive compilation and deployment of this code was impossible to the author. A question for a statement or help was answered with the comment that the project manager migrated to another university. Mails to his new university were completely ignored. The grid community has problems concerning code reuse itself [GCGS] which makes this approach even more confusing.

systems. With this analogy it is easy to understand that it is important for grid services to be interoperable (so the WS implementation is a step forward) but it does not aim to be a solution for a scientific middleware.

The next sections give a brief overview on available and important grid services.

7.2.5.1 Resource Management

Resource management for meta-computers deal with the problem of discovering and allocating computational resources, with authentication, process creation, and other activities to prepare a resource for use. A meta-computing environment introduces five challenging resource management problems: site autonomy, heterogeneous substrate, policy extensibility, co-allocation, and on-line control. The paper [CFK+98] proposes a solution for all these problems at once. The GLOBUS Resource Allocation Manager (GRAM) has evolved to be the most commonly used solution[4].

Two issues that GRAM does not address, is advanced reservation and heterogeneous resource types. The absence of advanced reservation means that GRAM cannot ensure that a resource provides a certain QoS. This drastically restricts the applicability of co-allocation on multiple sites. Specialised resources such as super-computers and high-bandwidth virtual channels are typically in high demand; reservation is of high importance. The lack of support for network, disk, and other resource types makes it impossible to provide end-to-end QoS guarantees if an application involves more than just closed computations. The article [FKL+99] presents an architectural approach to overcome these problems.

7.2.5.2 Data Management

File-transfer and therefore the integration of storage systems is an important issue for any computational environment. Grid-FTP, see `http://www.globus.org/toolkit/docs/4.0`.

7.2.5.3 Information Services

Information services give all grid components a common meta data basis. The MDS is a LDAP like directory service for the grid [CFFK01].

[4]Though inter site-communication and firewall problems are ignored; two components co-allocated on two different clusters cannot in general communicate directly.

Chapter 8

Motivation

Software components and parallel computing are topics of the previous parts of this article. But as mentioned before: scientific computing is an intrinsically interdisciplinary field and purely software-oriented research can hardly be applied directly in development of scientific software. This is the reason why scientific contribution in this field can hardly be made with purely software-oriented research. Dissertations as [Ber00] and research-projects as the CTL [Nie07a] show, that it is possible to contribute by inventing reusable generic architectures and tools for special applications, which can be reused by domain-experts. Such abstract frameworks – like modern template libraries in general [CE00] – encapsulate knowledge about a problem-domain in a reusable way. For a contribution like this, an domain of knowledge has to be analysed and engineered [CE00].

A very interesting field for software reuse is *partitioned analysis*. Partitioned analysis aims at reuse of available simulation software for coupled simulation of physically coupled phenomena. Vision is a generative component architecture for coupled analysis with following properties:

performance assumed to be a critical goal

- Configuring certain algorithm-features at compile-time
- Complete communication should be implemented using high performance MPI communication layer

flexibility Configuring certain algorithm-features at runtime using CTL mechanisms

extensibility To be reusable, extensibility has to be featured

- Component encapsulation must not be injured
- Interfaces have to be standardised and extensible at the same time

documentation variation points have to be documented

easy to use the interface to the library has to be simple enough

The rest of this part will be an overview of coupling algorithms for partitioned analysis. It has to be refined in future work to become a domain analysis proposed in [CE00].

Chapter 9

Terminology

The articles [HDGB99, FPF99] give an overview about basic terms in the terminology of coupled field problems. Terminology is very sensitive to changes, therefore statements are carefully adapted or completely taken without change. The most relevant terms for this part will be reviewed as follows:

field problem Subsystems are called physical fields when their mathematical model is described by field equations. Examples are mechanical and non-mechanical objects treated by continuum theories: solids, fluids, heat, electromagnetics [FPF99].

artificial subsystems Sometimes artificial subsystems are incorporated for computational convenience. Two examples: dynamic fluid meshes to effect volume mapping of Lagrangian to Eulerian descriptions in interaction of structures with fluid flow, and fictitious interface fields, often called “frames”, that facilitate information transfer between two subsystems [FPF99].

coupled system A coupled system or formulation is defined on multiple domains, possibly coinciding, involving dependent variables that cannot be eliminated on equation level [HDGB99]. It is a system in which physically or computationally heterogeneous mechanical components interact dynamically. The interaction is multiway in the sense that response has to be obtained by solving simultaneously the coupled system of equations.

“Heterogeneity” is used in the sense that a partition benefits from custom treatment [FPF99].

strong/weak physical interaction Very often a coupled system is called either **weak** or **strong coupled**. In a physical sense, **strong coupling** is present in coupled processes that can not be treated separately; i.e. strongly coupled processes interact mutually. **Weak coupling** is present in coupled processes that are separable [HDGB99], the interaction can be considered (approximately) as one-way.

A wind-turbine/aircraft/building interacts with fast moving air in its surrounding. The strong physical coupling appears in excited vibration and exerted forces in the structure induced by flow of gas in its surrounding. It appears in the gas in accustic waves (sound), a transition from laminar to turbulent flow and other phenomena. A weak physical coupling can be observed if the interaction of two phenomena is strong only in one-direction and negligible in the other direction.

There is an obvious problem; if coupled problems are studied, it is often not very good known how strong, respectively weak they are physically coupled; this is an answer expected from the simulation of the overall problem. If material properties describing parameters are non-linearly depending on the field quantities, the coupling (strong/weak) can change with varying field quantities and the field quantities result from the analysis. Therefore,

the definition of strong/weak coupling should be chosen with respect to the numerical aspects instead to their physical nature [HDGB99].

Choosing for numerical aspects, it is possible to have a combined strong/weak coupling of field problems. This means that the strategy of coupling can vary during the solution process [HDGB99].

Numerical Strong Coupling Numerical strong coupling is a full coupling of the model equations on algebraic level. The equations of all involved and modeled effects are solved simultaneously with respect to full mutual interaction. That implies, that coupling terms are introduced as algebraic constraints. [HDGB99].

Numerical Weak Coupling Numerical weak coupling means that considered field partitions are solved successively. Coupling is introduced by solving for the state of a partition isolated from dependent partitions and transferring field properties to dependent fields which are then solved dependently [HDGB99].

Monolithic Approach In the so-called monolithic approach to coupled simulation a new software (or solution method) is developed for each coupling [MNS06]; i.e. the resulting monolitic software system solves a unique coupled problem.

Partitioned Approach In the so-called partitioned approach to coupled simulation a (strong or weak) coupling method is used for introducing the coupling of a set of partitions. It is possible to reuse available software and optimised methods for the partitions.

n-Field Decomposition A coupled system is characterised as multi-field problem (i.e. two-field, three-field, etc.) according to the number of different fields that appear in the first-level decomposition [FPF99]. This fields can be artificial subsystems like interface frames or models of physical fields.

9.1 Routines of Partitioned Analysis

Felippa et al. [FPF99] give an overview of some standard routines which are frequently used in partitioned analysis:

Prediction a predictor is used to *guess* the next state (or the next interface state) of an adjacent system partition

Data Exchange transferring state information from one partition to another

Interfield Iteration iteratively solve coupled partitions until a stationary state is reached

Full step correction transferring state-information from an evolved partition to a non-evolved partition

Lockstep advancing evolving the state of partitions independently (no coupling)

Midpoint correction evolving partitions in different time-scales and transferring state-information from an evolved partition state back to a less evolved partition

Subcycling evolving a partition on finer time-scale than the other

Translation convert quantities of a partition to related quantities of another partition

9.2 Coupled Simulation

In the modelling of physical systems conservation laws like *conservation of energy*, *mass*, *angular momentum*, *electric charge* or *probability* play an important role. Closed natural systems satisfy these laws naturally, whereas they have to be modeled explicitly in a computer simulation to capture a real world behaviour. Coupled systems have common quantities on which conservation has to be adequately handled (e.g. transfer or reflection). This kind of conservative transfer is the main effort of coupled simulation. Coupled analysis has many routines in common with contact mechanics and domain decomposition, the main differences are in the aim of the methods. These aims have some implications on the applicability of a method in a domain.

Contact Mechanics aims at the simulation of areas under contact or impact (contact with positive approximation velocity)

- relevant goals
  - modelling realistic physical behaviour [Wri02]
  - consistent results [Wri02]
- dispensable goals
  - computational efficiency

Coupled Analysis aims at the simulation of long term coupled evolution processes on independently handled spatial domains

- relevant goals
  - reuse of available efficient simulation components [FPF99, MNS06]
  - consistent results [FPF99, MNS06]
- dispensable goals
  - fast convergence properties
  - computational efficiency
  - minimising of partition communication

Domain Decomposition aims at the spatial structure decomposition for parallel computing or multigrid methods

- relevant goals
  - computational efficiency [SBG96]
  - same results as the original non-decomposed system [SBG96]
  - minimising of partition communication
  - fast convergence properties [SBG96]
- dispensable goals
  - reuse of available efficient simulation components

Methods and algorithms for one of the listed domains get frequently modified to also fulfil the requirements of other fields. A common proceeding in all three domains is the *method of Lagrange multipliers*, which generally aims at non-linear constrained optimisation. Another routine, the *penalty method*, introduces stiff coupling springs on the contact area and therefore introduces non-physical penetration [Wri02][1].

[1] The stiff springs also change the condition of the resulting system which is one of the major disadvantage of this Method [Wri02]

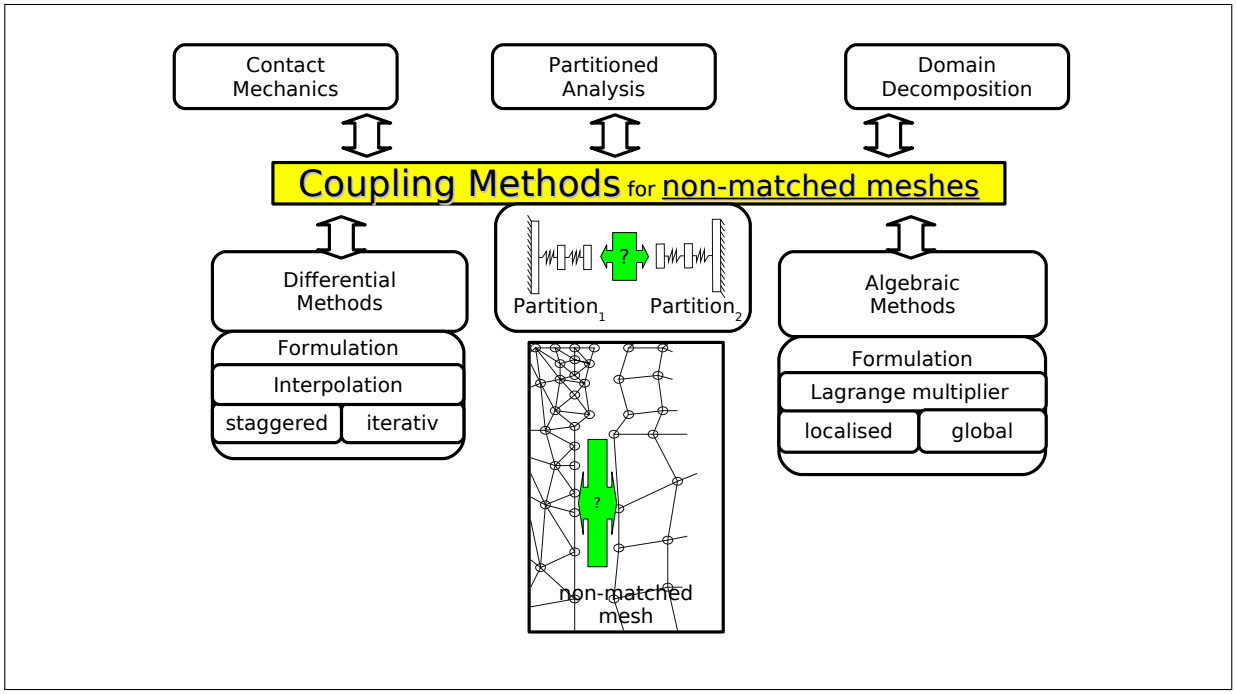

Figure 9.1: *The picture gives an overview of known approaches for the coupling of partitions with non-matched meshes. The differential approaches are based on interpolation, while algebraic methods are based on a finite element ansatz for the energy on the interface; i.e. the total energy in the system (including the energy on the interface) is to be minimised.*

Chapter 10

Partitioned Analysis

Complex systems always need to be decomposed and classified in order to be understandable, they are amenable to many kinds of decomposition according to specific analysis or design goals. Partitioned simulations evolute the state of a partition under specified conditions viewed as external to the partition.

The term *partitioned analysis* is used in the literature as synonym for the methods behind the partitioned approach to coupled simulation systems. The term *partitioned approach* describes the category of coupling algorithms, which are made for the subsequent coupling of monolithic simulators, which are not designed to be coupled. These monolithic simulators do not implement necessary features, which are required to make the solver out of the box combinable with other simulators.

Felippa et al. [FPF99] introduce a hierarchy of partitioning approaches as follows: The term partitioning identifies the process of spatial separation of a discrete model into interacting components called partitions. The decomposition may be driven by physical, functional, or computational considerations. Substructures can be further decomposed into submeshes or subdomains to accommodate parallel computing requirements. Subdomains are composed of individual elements. This kind of multilevel partition hierarchy: coupled system $\triangleright$ structure $\triangleright$ substructure $\triangleright$ subdomain $\triangleright$ element, is part of current modelling and solution proceeding.

Non-matched interface meshes are not a special case, but are the general and standard case in multi-physic and multi-scale simulation. Two major approaches for this problem exist, which have to be examined here. Partitioning may be *algebraic* or *differential*. *Differential partitioning* methods aim at minimising the difference between the shared variables, and the shared variables are redundantly defined local in both partitions. To introduce a common state, the shared state-variables have to be transfered to the according adjacent partition at least one-time (staggered method) or until convergence is achieved (iterative method). In the process of partition-communication (state transfer) on non-matched meshes remote data generally has to be projected onto the local interface nodes. This can be done in two ways; the most convenient one is to evaluate the remote ansatz at the local nodes. If this is not implemented in the given remote solver then the values have to be interpolated extrinsically through an appropriate interpolation. *Algebraic partitioning* aims at coupling the given partitions through algebraic constrains. These constrains need to be minimised. This minimisation process can be introduced through Lagrange multiplier fields or penalty factors. Lagrange multipliers are better applicable than penalty factors since they do not introduce non-physical behaviour as large penalty factors do.

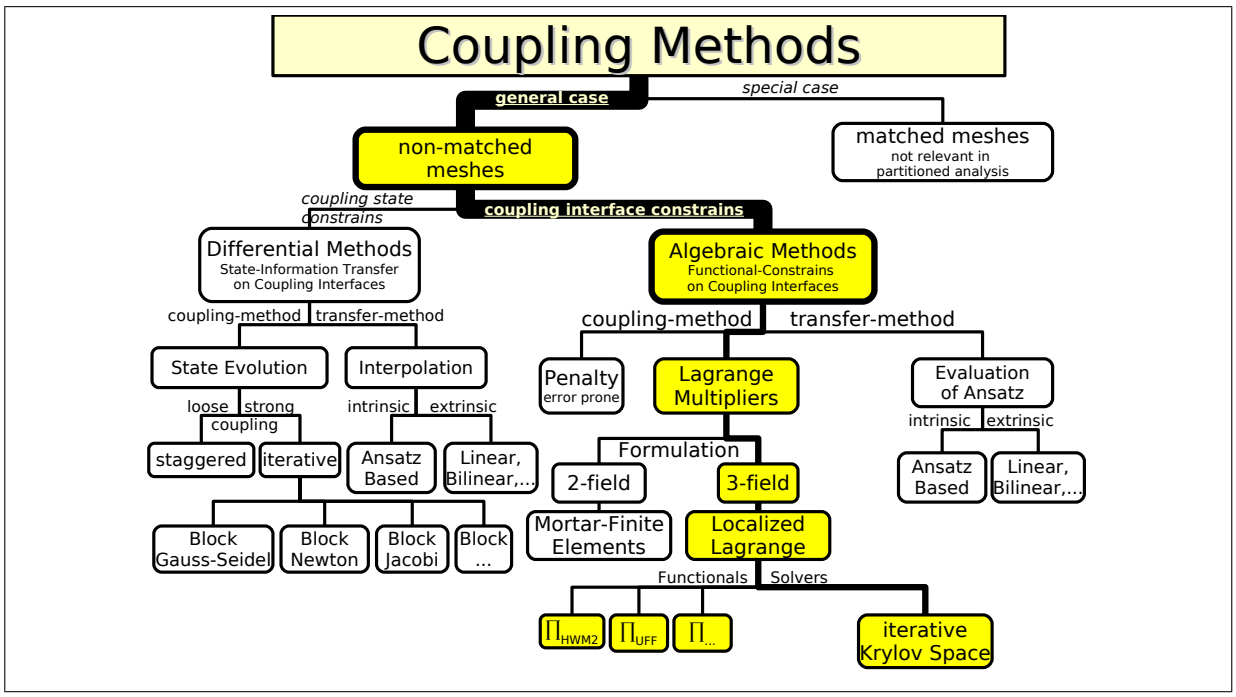

Figure 10.1: *The picture gives an overview of known coupling methods. Only coupling methods for non-matching interfaces are relevant for partitioned analysis. Differential approaches (in the left sub-tree) also lead to strongly coupled systems, but not necessarily [MNS06]. Here the common interface state evolves iteratively through mutual interface state transfers and repeated partition evaluations. The path marked in yellow is the one which is chosen for consolidation here. In algebraic methods, the state of the partitioned system evolves strongly coupled. The system state transition is modelled by a global system of equations which is then solved by iterative methods which allows the reuse of solvers for system partitions.*

Chapter 11

Lagrange Multiplier Methods

In contact problem formulations there is the general choice between different methods for the treatment of contact problems, including the Lagrange multiplier or the penalty formulation. Both formulations are formulated in *strong form* meaning on the algebraic equation-level.

The common goal of these approaches is to minimise the deviations between values, which are adjacent across the coupling interface on a coupled boundary. The Lagrange multiplier method introduces additional variables in the system, but does fulfil the constraint equation correctly. The penalty formulation in contrast leads to observable non-physical penetration but does not need additional variables [Wri02]. Non-physical treatment is unacceptable for coupled analysis, where results have to be equal to the results of a monolithic solver — meaning where strong coupling is the goal to be achieved. The penalty method is therefore not applicable in partitioned analysis.

The Lagrange-based approaches can be divided into two categories; one of both categories contains formulations leading to semi-partitioned solvers (see section 11.2) and the other one leads to a partitioned approach (see section 11.3) — namely an approach for the coupling of multiple independently developed solvers. Both Lagrange-based approaches have different properties which are to be listed and analysed; for example the number of introduced unknowns.

11.1 Derivation of Coupling with Lagrangian Multiplier

This Section introduces examples for the different application of Lagrange multipliers in the coupling of partitioned systems. The first simple example couples two independently modeled spring-mass systems with both a *localised* and a *global* multiplier. The second part summarises results for non-matching meshes.

11.1.1 Simple Example

Consider the simple one-dimensional mass-spring systems S_1 and S_2 in figure 11.1. The system coupling in this example will be considered in three ways; in this section we derive a monolithic solver, a global Lagrange solver and a localised Lagrange solver for the given system. Let us start with the simplest in this scenario — the monolithic solver.

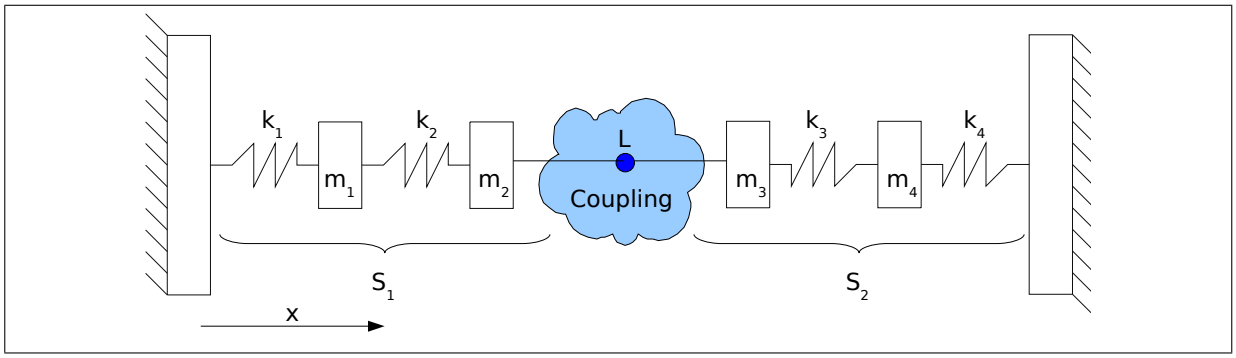

Figure 11.1: *The picture shows a simple partitioned system which is to be coupled.*

Monolithic Solver

The system is coupled at the join point L; the coupled system can be reduced to a monolithic system by joining the masses m_2 and m_3 to a virtual mass $m_{2,3} = m_2 + m_3$ and writing down the energy equilibrium equation.

$$\frac{1}{2}\left[k_1 u_1^2 + k_2 \left(u_{2,3} - u_1\right)^2 + k_3 \left(u_4 - u_{2,3}\right)^2 + k_4 u_4^2\right] - f_1 u_1 - f_{2,3} u_{2,3} - f_4 u_4 = \Pi(u_1, u_{2,3}, u_4). \tag{11.1}$$

We are searching for the stationary solution which can be found by solving for the root of the first derivative. Writing down the variations with respect to the unknowns $u_1, u_{2,3}, u_4$ yields

$$\frac{\delta \Pi}{\delta u_1} = (k_1 + k_2)u_1 - k_2 u_{2,3} - f_1 = 0$$

$$\frac{\delta \Pi}{\delta u_{2,3}} = -k_2 u_1 + (k_2 + k_3)u_{2,3} - k_3 u_4 - f_{2,3} = 0$$

$$\frac{\delta \Pi}{\delta u_4} = (k_3 + k_4)u_4 - k_3 u_{2,3} - f_4 = 0.$$

This systems can be represented in matrix-form as

$$\begin{pmatrix} (k_1 + k_2) & -k_2 & 0 \\ -k_2 & (k_2 + k_3) & -k_3 \\ 0 & -k_3 & (k_3 + k_4) \end{pmatrix} \begin{pmatrix} u_1 \\ u_{2,3} \\ u_4 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_{2,3} \\ f_4 \end{pmatrix}.$$

In this linear system of equations $\mathbf{A}\boldsymbol{u} = \boldsymbol{f}$ the vector $\boldsymbol{u}$ is the unknown vector of displacements under external forces $\boldsymbol{f}$, which depend on the mass vector $\boldsymbol{m} = (m_1, m_{2,3}, m_4)^T$ and may be chosen by introducing gravity[1] $\boldsymbol{f} = 9.81 * \boldsymbol{m}$. The solution must be interpreted to become the same as the coupled solutions in which the masses m_2 and m_3 do not need to be joined, but the resulting displacements u_2 and u_3 are the same. The solution of the monolithic solver is therefore the vector $\boldsymbol{u} = (u_1, u_{2,3}, u_{2,3}, u_4)^T$.

Global Lagrange Solver

Another way to couple the two independent systems S_1 and S_2 in figure 11.1 is to introduce an algebraic coupling constraint into the energy equilibrium equation 11.1; here both displacements u_2 and u_3 of the masses m_2 and m_3 should be the same in the final solution. The system S_1 is modelled independently in equation 11.2.

$$\frac{1}{2}\left[k_1 u_1^2 + k_2 \left(u_2 - u_1\right)^2\right] - f_1 u_1 - f_2 u_2 = \Pi_1(u_1, u_2). \tag{11.2}$$

[1] in x direction of figure 11.1

This models a system with two masses and two springs. Again we are searching for the stationary solution which can be found by solving for the root of the first derivative. Writing down the variations with respect to the unknowns u_1, u_2 yields

$$\frac{\delta\Pi}{\delta u_1} = (k_1 + k_2)u_1 - k_2 u_2 - f_1 = 0$$
$$\frac{\delta\Pi}{\delta u_2} = -k_2 u_1 + k_2 u_2 - f_2 = 0$$

Which again can be written in matrix-form:

$$\begin{pmatrix} (k_1+k_2) & -k_2 \\ -k_2 & k_2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}.$$

The system matrix for $\mathbf{S}_2$ can be derived in the same way and yields:

$$\begin{pmatrix} k_3 & -k_3 \\ -k_3 & (k_4+k_3) \end{pmatrix} \begin{pmatrix} u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} f_3 \\ f_4 \end{pmatrix}.$$

The coupling constraint can now be introduced by defining $u_2 = u_3$ meaning the displacements of the two masses are now forced to be the same. This constraint can be formulated by introducing a Lagrange multiplier and rewriting the constrain as $\lambda(u_2 - u_3) = 0$. This term will additionally be introduced into to energy equilibrium function $\Pi_1(u_1, u_2) + \Pi_2(u_3, u_4) + \lambda(u_2 - u_3) = \Pi_{Glo}$ and yields the coupled system:

$$\begin{pmatrix} (k_1+k_2) & -k_2 & 0 & 0 & 0 \\ -k_2 & k_2 & 0 & 0 & 1 \\ 0 & 0 & k_3 & -k_3 & -1 \\ 0 & 0 & -k_3 & (k_4+k_3) & 0 \\ 0 & 1 & -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \lambda \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ 0 \end{pmatrix}.$$

Observation: The matrix has the following general block-structure:

$$\begin{pmatrix} \mathbf{A}_1 & \mathbf{0} & \mathbf{B}_1 \\ \mathbf{0} & \mathbf{A}_2 & -\mathbf{B}_2 \\ \mathbf{B}_1^T & -\mathbf{B}_2^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}_1 \\ \boldsymbol{f}_2 \\ 0 \end{pmatrix}$$

Where $\mathbf{A}_1$ is the system matrix of system $\mathbf{S}_1$, $\mathbf{A}_2$ is the system matrix of system $\mathbf{S}_2$, $\mathbf{B}_1$ and $\mathbf{B}_2$ are the discretised coupling term. The solution is exactly the same as the solution of the monolithic solver.

Localized Lagrange Solver

This section follows the description in [PFO04]. In localized Lagrange the system matrices $\mathbf{A}_1$ and $\mathbf{A}_2$ are the same as in the global Lagrange solution described in the last section. The difference is in the formulation of the coupling constraint, which is formulated with respect to an artificially introduced degree of freedom u_I. Here the coupling constraint is formulated in two steps $\lambda_1(u_2 - u_I) = 0$ and $\lambda_2(u_3 - u_I) = 0$, which yields $\Pi_1(u_1, u_2) + \Pi_2(u_3, u_4) + \lambda_1(u_2 - u_I) + \lambda_2(u_3 - u_I) = \Pi_{LLM}$ and results in the fairly big system of equations

$$\begin{pmatrix} (k_1+k_2) & -k_2 & 0 & 0 & 0 & 0 & 0 \\ -k_2 & k_2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & k_3 & -k_3 & 0 & 1 & 0 \\ 0 & 0 & -k_3 & (k_4+k_3) & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \lambda_1 \\ \lambda_2 \\ u_I \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Observation: The matrix has the following general block-structure [PFO04]:

$$\begin{pmatrix} \mathbf{A}_1 & \mathbf{0} & \mathbf{B}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_2 & \mathbf{0} & \mathbf{B}_2 & \mathbf{0} \\ \mathbf{B}_1^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & -\mathbf{L}_1 \\ \mathbf{0} & \mathbf{B}_2^T & \mathbf{0} & \mathbf{0} & -\mathbf{L}_2 \\ \mathbf{0} & \mathbf{0} & -\mathbf{L}_1^T & -\mathbf{L}_2^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{u}_{(1)} \\ \boldsymbol{u}_{(2)} \\ \boldsymbol{\lambda}_{(1)} \\ \boldsymbol{\lambda}_{(2)} \\ \boldsymbol{u}_I \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}_{(1)} \\ \boldsymbol{f}_{(2)} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}$$

Where $\mathbf{A}_1$ is the system matrix of system $\mathbf{S}_1$, $\mathbf{A}_2$ is the system matrix of system $\mathbf{S}_2$. $\mathbf{B}_1$, $\mathbf{B}_2$, $\mathbf{L}_1$, $\mathbf{L}_2$ are the discretised coupling terms. The solution is exactly the same as the solution of the monolithic solver.

Comparing the Results

The last section 11.1.1 executed the relevant methods on a very simple example. This simple example already yields some advantages and disadvantages of the given methods:

Method	**Monolithic**	**Global Lagrange**	**Localized Lagrange**
non-intrusive	–	$\surd$	$\surd$
additional complexity	0	n DOFs on Γ_I	$3n$ DOF on Γ_I
modelling complexity	high	low	low
reuse of partitions	–	$\surd$	$\surd$

As the table shows both global and localised Lagrange multipliers are non-intrusive in this example, this results to be the case also in the general case, which will be demonstrated later. One can furthermore see from the example, that the solvers for the subsystems S_1 and S_2 namely A_1^{-1} and A_2^{-1}, respectively can for example be used as black-box proconditioners for a iterative Krylov-space solver. The example hides the software technical advantages of the localised Lagrange method, which is its openness, since it is possible to couple more than two partitions like in figure 11.2 without changing a single solver.

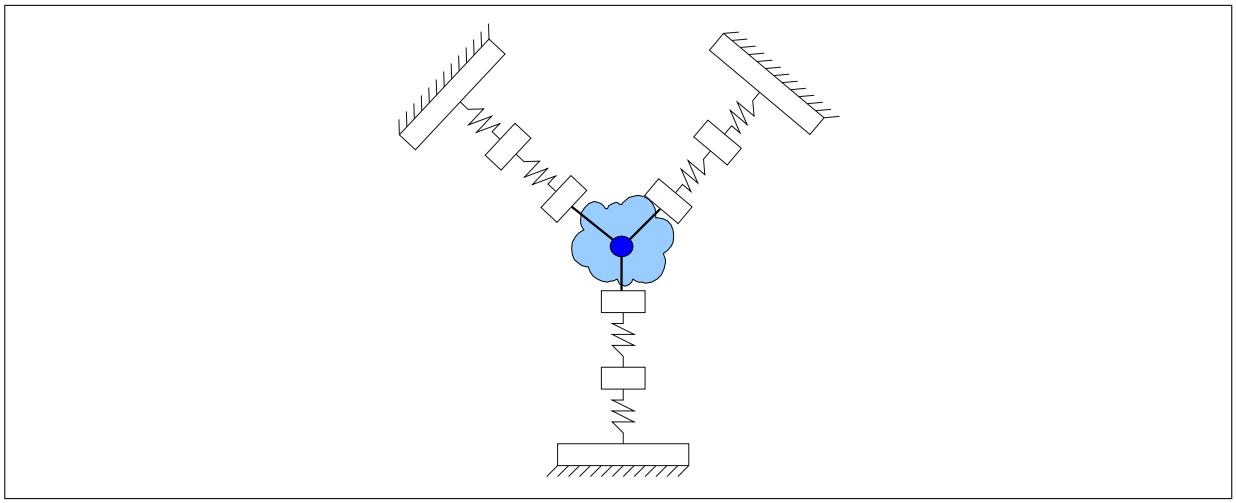

Figure 11.2: *The picture shows a situation in which three partitions are to be coupled. This situation can be handled with localized Lagrange without changes in the solvers of the given partitions .*

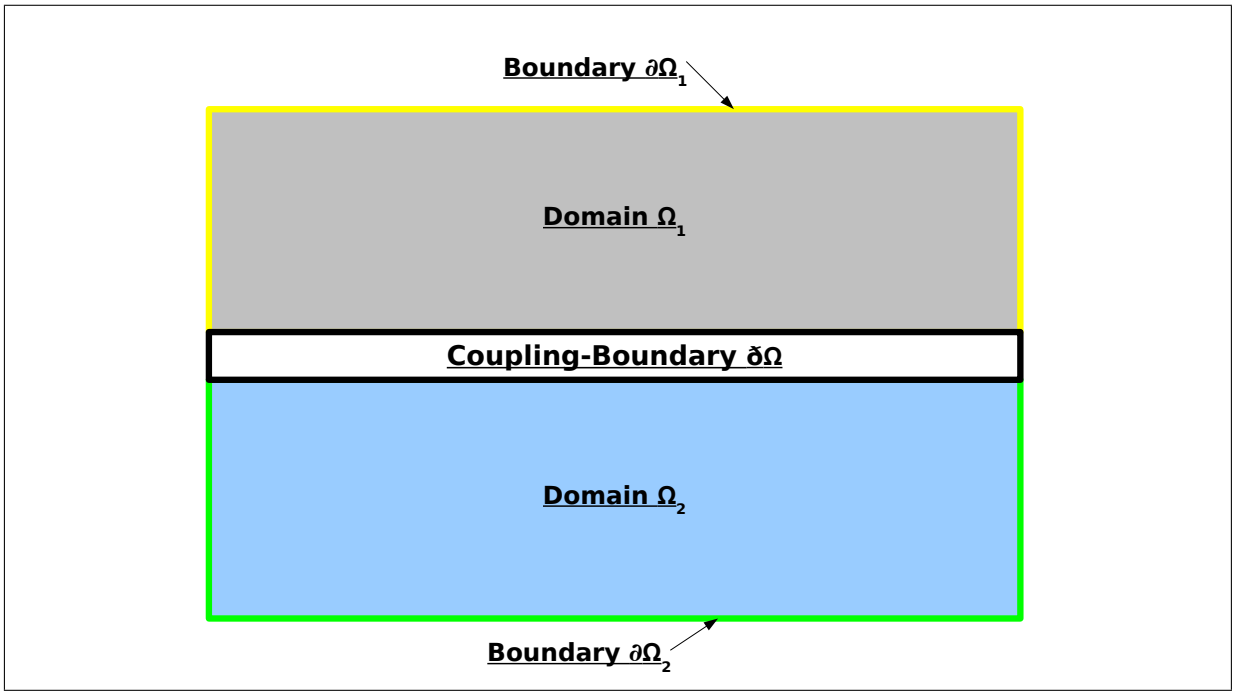

Figure 11.3: *An example partitioned domain.*

11.1.2 Coupling of Complex Partitioned Systems

Consider the partitioned domain in figure 11.3 with $\Omega = \Omega_1 \cup \Omega_2$ and $\eth\Omega = \delta\Omega_1 \cap \delta\Omega_2$. For the composed system shown the model may be stated as [PFO04]:

$$\delta\Pi\left(\boldsymbol{u}_g\right) = \delta\boldsymbol{u}_g^T(\mathbf{A}_g(\boldsymbol{u}_g) - \boldsymbol{f}_g) \tag{11.3}$$

Where $\mathbf{A}_g$ is the global assembled discrete model, $\boldsymbol{u}_g$ is the global vector of unknowns, $\boldsymbol{f}_g$ is the external generalised force acting on the assembled system. The coupled system can now be considered to be composed of independent fields connected through interface constrains. The common partition boundary $\eth\Omega$ can be treated by the methods of Lagrange multipliers demonstrated in the last sections 11.1.1. The nodes of $\eth\Omega$ are redundantly introduced in both fields Ω_1 and Ω_2 and therefore vectors $\boldsymbol{u}_{\eth\Omega\setminus\Omega_2}$ and $\boldsymbol{u}_{\eth\Omega\setminus\Omega_1}$ are defined in both partitions, whereas the two common boundary displacements should be approximately the same in a solution. In the following sub-sections the solution approaches will be reviewed summarising the work of K.C. Park et al. [PFO04, PF00, FPF99, PFR01, PFG00, PFR02, GPF00].

11.1.2.1 Global Lagrange Approach

As demonstrated with the simple examples in section 11.1.1, the coupling constraint can be expressed as:

$$\boldsymbol{u}_{\eth\Omega\setminus\Omega_2} - \boldsymbol{u}_{\eth\Omega\setminus\Omega_1} = 0 \tag{11.4}$$

or more generally with an arbitrary coupling functional g:

$$g(\boldsymbol{u}_{\eth\Omega\setminus\Omega_2}, \boldsymbol{u}_{\eth\Omega\setminus\Omega_1}) = 0 \tag{11.5}$$

We will assume here as in [PFO04] that each of the subsystems and the interface constrains can be modeled as

$$\begin{array}{ll} \text{Subsystem 1:} & \delta\Pi_{(1)} = \delta\boldsymbol{u}_{(1)}^T \left[\mathbf{A}_{(1)}(\boldsymbol{u}_{(1)}) - \boldsymbol{f}_{(1)}\right] \\ \text{Subsystem 2:} & \delta\Pi_{(2)} = \delta\boldsymbol{u}_{(2)}^T \left[\mathbf{A}_{(2)}(\boldsymbol{u}_{(2)}) - \boldsymbol{f}_{(2)}\right] \\ \text{Interface functional:} & \pi_{global_\lambda} = \left[\boldsymbol{\lambda}_{(12)} g(\boldsymbol{u}_{\partial\Omega\setminus\Omega_2}, \boldsymbol{u}_{\partial\Omega\setminus\Omega_1})\right] \end{array} \tag{11.6}$$

where $\mathbf{A}_{(1)}$ and $\mathbf{A}_{(2)}$ are the partition-level governing operators and $\boldsymbol{u}_{(1)}$ and $\boldsymbol{u}_{(2)}$ are the degrees of freedom of the two subsystems. The constrained degrees of freedom of the two partitions are part of the degrees of freedom $\boldsymbol{u}_{(1)}$ and $\boldsymbol{u}_{(2)}$. Therefore the constrained variables $\boldsymbol{u}_{\partial\Omega\setminus\Omega_2}$ and $\boldsymbol{u}_{\partial\Omega\setminus\Omega_1}$ can be expressed in terms of $\boldsymbol{u}_{(1)}$ and $\boldsymbol{u}_{(2)}$ as

$$\begin{aligned} \boldsymbol{u}_{\partial\Omega\setminus\Omega_2} &= \mathbf{B}_{(1)}\boldsymbol{u}_{(1)} \\ \boldsymbol{u}_{\partial\Omega\setminus\Omega_1} &= \mathbf{B}_{(2)}\boldsymbol{u}_{(2)} \end{aligned} \tag{11.7}$$

where $\mathbf{B}^{(1)}$ and $\mathbf{B}^{(2)}$ are Boolean matrices that extract only the interface degrees of freedom at each subsystem interfaces.

The variational functional which models the total system can be written as:

$$\begin{aligned} \delta\Pi_{total} &= \delta\Pi_{(1)} + \delta\Pi_{(2)} + \pi_{global_\lambda} \\ &= \delta\boldsymbol{u}_{(1)}^T \left[\mathbf{A}_{(1)}(\boldsymbol{u}_{(1)}) - \boldsymbol{f}_{(1)}\right] + \delta\boldsymbol{u}_{(2)}^T \left[\mathbf{A}_{(2)}(\boldsymbol{u}_{(2)}) - \boldsymbol{f}_{(2)}\right] \\ &\quad + \left[\boldsymbol{\lambda}_{(12)} g(\boldsymbol{u}_{\partial\Omega\setminus\Omega_2}, \boldsymbol{u}_{\partial\Omega\setminus\Omega_1})\right] \end{aligned} \tag{11.8}$$

which by using $\mathbf{B}^{(1)}$ and $\mathbf{B}^{(2)}$ can be rewritten as:

$$\begin{aligned} \delta\Pi_{total} &= \delta\boldsymbol{u}_{(1)}^T \left[\mathbf{A}_{(1)}(\boldsymbol{u}_{(1)}) - \boldsymbol{f}_{(1)}\right] + \delta\boldsymbol{u}_{(2)}^T \left[\mathbf{A}_{(2)}(\boldsymbol{u}_{(2)}) - \boldsymbol{f}_{(2)}\right] \\ &\quad + \left[\boldsymbol{\lambda}_{(12)} g(\mathbf{B}_{(1)}\boldsymbol{u}_{(1)}, \mathbf{B}_{(2)}\boldsymbol{u}_{(2)})\right] \end{aligned} \tag{11.9}$$

The stationary $\delta\Pi_{total} = 0$ of the variational expression (11.9) leads to the following partitioned system of equations:

$$\begin{pmatrix} \mathbf{A}_{(1)} & 0 & \mathbf{B}_{(1)} \\ 0 & \mathbf{A}_{(2)} & -\mathbf{B}_{(2)} \\ \mathbf{B}_{(1)}^T & -\mathbf{B}_{(2)}^T & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{u}_{(1)} \\ \boldsymbol{u}_{(2)} \\ \boldsymbol{\lambda}_{(12)} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}_{(1)} \\ \boldsymbol{f}_{(2)} \\ 0 \end{pmatrix}$$

which results to be the same as (11.1.1) on page 61. It should be noted, that the Lagrange multiplier $\boldsymbol{\lambda}_{(12)}$ is a global variable.

11.1.2.2 Localized Lagrange Approach

In analogy to the simple one-dimensional examples in section 11.1.1 and the descriptions in the last section, we can formulate the localised approach respectively with the following modified constrains:

$$\begin{aligned} \boldsymbol{u}_{\partial\Omega\setminus\Omega_2} - \boldsymbol{u}_I &= 0 \\ \boldsymbol{u}_{\partial\Omega\setminus\Omega_1} - \boldsymbol{u}_I &= 0 \end{aligned} \tag{11.10}$$

or again more generally with an arbitrary coupling functional g:

$$\begin{aligned} g(\boldsymbol{u}_{\partial\Omega\setminus\Omega_2}, \boldsymbol{u}_I) &= 0 \\ g(\boldsymbol{u}_{\partial\Omega\setminus\Omega_1}, \boldsymbol{u}_I) &= 0 \end{aligned} \tag{11.11}$$

We follow the describtion in [PFO04] assuming that each of the subsystems and the interface constrains can be modeled as follows:

$$
\begin{array}{lll}
\text{Subsystem 1:} & \delta\Pi_{(1)} = \delta \boldsymbol{u}_{(1)}^T \left[\mathbf{A}_{(1)}(\boldsymbol{u}_{(1)}) - \boldsymbol{f}_{(1)}\right] \\
\text{Subsystem 2:} & \delta\Pi_{(2)} = \delta \boldsymbol{u}_{(2)}^T \left[\mathbf{A}_{(2)}(\boldsymbol{u}_{(2)}) - \boldsymbol{f}_{(2)}\right] \\
\text{Interface functional:} & \pi_{local_\lambda} = \left[\boldsymbol{\lambda}_{(1)} g_1(\boldsymbol{u}_{\eth\Omega\setminus\Omega_2}, \boldsymbol{u}_I)\right] + \left[\boldsymbol{\lambda}_{(2)} g_2(\boldsymbol{u}_{\eth\Omega\setminus\Omega_1}, \boldsymbol{u}_I)\right]
\end{array}
\tag{11.12}
$$

which by can be rewritten using again the *Boolean filter matrices* $\mathbf{B}^{(1)}$ and $\mathbf{B}^{(2)}$ as:

$$
\begin{array}{rl}
\delta\Pi_{total} & = \delta \boldsymbol{u}_{(1)}^T \left[\mathbf{A}_{(1)}(\boldsymbol{u}_{(1)}) - \boldsymbol{f}_{(1)}\right] + \delta \boldsymbol{u}_{(2)}^T \left[\mathbf{A}_{(2)}(\boldsymbol{u}_{(2)}) - \boldsymbol{f}_{(2)}\right] \\
& + \left[\boldsymbol{\lambda}_{(1)} g_1(\mathbf{B}_{(1)}\boldsymbol{u}_{(1)}, \boldsymbol{u}_I)\right] + \left[\boldsymbol{\lambda}_{(2)} g_2(\mathbf{B}_{(2)}\boldsymbol{u}_{(2)}, \boldsymbol{u}_I)\right]
\end{array}
\tag{11.13}
$$

The stationary $\delta\Pi_{total} = 0$ of the variational expression (11.13) leads to the following partitioned system of equations:

$$
\begin{pmatrix}
\mathbf{A}_1 & \mathbf{0} & \mathbf{B}_1 & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{A}_2 & \mathbf{0} & \mathbf{B}_2 & \mathbf{0} \\
\mathbf{B}_1^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & -\mathbf{L}_1 \\
\mathbf{0} & \mathbf{B}_2^T & \mathbf{0} & \mathbf{0} & -\mathbf{L}_2 \\
\mathbf{0} & \mathbf{0} & -\mathbf{L}_1^T & -\mathbf{L}_2^T & \mathbf{0}
\end{pmatrix}
\begin{pmatrix}
\boldsymbol{u}_{(1)} \\ \boldsymbol{u}_{(2)} \\ \boldsymbol{\lambda}_{(1)} \\ \boldsymbol{\lambda}_{(2)} \\ \boldsymbol{u}_I
\end{pmatrix}
=
\begin{pmatrix}
\boldsymbol{f}_{(1)} \\ \boldsymbol{f}_{(2)} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0}
\end{pmatrix}
$$

11.1.2.3 Non-matching Meshes

The general case in coupling of partitioned systems is a non-matching mesh on the interface boundary $\eth\Omega$. In the Lagrangian approach this problem can be reduced to an integration over the interface:

$$
\pi_{global_\lambda} = \int_{\eth\Omega} \boldsymbol{\lambda}_{(12)}(x) \left\{ g(\boldsymbol{u}_{\eth\Omega\setminus\Omega_2}(\boldsymbol{x}), \boldsymbol{u}_{\eth\Omega\setminus\Omega_1}(\boldsymbol{x}) \right\} d\boldsymbol{x}
$$

or in localized Lagrange respectively:

$$
\pi_{local_\lambda} = \int_{\eth\Omega} \{ \left[\boldsymbol{\lambda}_{(1)}(\boldsymbol{x}) g_1(\boldsymbol{u}_{\eth\Omega\setminus\Omega_1}(\boldsymbol{x}), \boldsymbol{u}_I(\boldsymbol{x}))\right] + \left[\boldsymbol{\lambda}_{(2)}(\boldsymbol{x}) g_2(\boldsymbol{u}_{\eth\Omega\setminus\Omega_1}(\boldsymbol{x}), \boldsymbol{u}_I(\boldsymbol{x}))\right] \} d\boldsymbol{x}.
$$

The localised Lagrange multiplier field ($\boldsymbol{\lambda}_{(1)}$,$\boldsymbol{\lambda}_{(2)}$) on both sides of the common artificial interface frame has important advantages over the global approach. While the global formulation enforced the coupling functional, (g), to be formulated with respect to both partitions at the same time, allows the localised formulation to formulate the coupling constrains (g_1,g_2) independently.

11.2 Semi-Partitioned Approach with Lagrange Multipliers

The popular and most confirmed approach to the problem of coupling of non-matching meshes is the *Mortar Finite Element Method* (MFEM) [Woh01]. This method was also applied to multi-physic applications in dozens of publications by Wohlmuth et al (e.g. [LW04]). The Book [Woh01] by Barbara Wohlmuth presents the mathematical foundation of the method. In some sense the method is a partitioned approach, since it provides the possibility to develop solvers, which can be coupled out of the box. The limit of this approach is the coupling of more than two phenomena like illustrated in figure 11.2 on page 62, which is the reason why we classify this approach as a semi-partitioned one.

The MFEM approach introduces a (global) field of Lagrange multipliers between two partitions and discretises the field with a finite element ansatz, which gets located in one of the both partitions — namely the mortar side [Woh01]. This approach introduces a non-symmetric situation in making one partition a *master* and the other

partition a *slave*. A solver implementation would therefore implement the whole coupling constrains locally inside one partition. The assembly of the FEM system matrix on the mortar-side further needs to integrate a remote ansatz function from the non-mortar side.

11.3 Partitioned Approach with Lagrange Multipliers

Coupling approaches with their focus on reuse are staggered and iterative methods as well as the newer approach proposed by Felippa, Park, Farhat [PF00] called *Localized Lagrange Method*. Staggered and iterative methods were examined in many publications for example in [MNS06, Ste04]. The Localized Lagrange Method is an alternative to these approaches and is related to the mortar finite element method, since both minimise the energy on the interface using Lagrange multipliers.

The difference between the Mortar Element method *with global Lagrange multipliers [PF00]* and the localised method is the introduction of an artificial interface frame between the coupling partitions. The given algebraic interface constrains can now on both sides be formulated with respect to partition-independent degrees of freedom on the artificial interface frame. This approach introduces a symmetric situation in making both partitionens a slave of the artificial interface. It should be remarked that this method does not yield any advantages from the mathematical point of view since it can easily be reduced to the mortar approach as demonstrated in [PFO04], but additionally even introduces redundancy. From a software-architectural point of view this solution yields a better decoupling of solvers of coupling partitions [PFO04]. A solver implementation would therefore implement no coupling locally inside its partition as it is done in the MFEM. The assembly of the FEM system matrix in the partitions does not even depend on the ansatz-functions of the interface-frame, since the complete interpolation work can be done on the artificial boundary and then only needs to be transfered onto the boundary of the given partition.

The localised Lagrange approach introduces redundancy into the algebraic coupling constrains. The constrains get formulated with respect to a artificially introduced variable — the interface frame. This is done by introducing one degree of freedom for the displacement of the frame which in the solution has to be the same as the displacement of the mutual partitions. This redundancy causes the localised Lagrange method to introduce three times as many degrees of freedom as the mortar-method does. The new degrees of freedom on the interface-frame are the anchor point for mutual Lagrange-multiplier fields connecting the partitions which are to be coupled to a common variable — namely the interface-frame. This does not need to be a single partition, as it is in the MFEM approach, but arbitrary many partitions can be coupled through one interface frame.

Glossary

AltiVec	Floating point and integer SIMD instruction set designed and owned by Apple, IBM and Freescale Semiconductor, formerly the semiconductor products sector of Motorola, and implemented on versions of the PowerPC including Motorola's G4, IBM's G5 and POWER6 processors, and P.A. Semi's PWRficient PA6T. See `http://en.wikipedia.org/wiki/AltiVec`, 36
ALU	**A**rithmetic **L**ogical **U**nit, 34
API	An *application programming interface* (API) is a source code interface that a computer application, operating system or library provides to support requests for services. See `http://en.wikipedia.org/wiki/API`., 28
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange, is a character encoding based on the English alphabet. Work on ASCII began in 1960. See `http://en.wikipedia.org/wiki/ASCII`, 43
CA	**C**ertification **A**uthority, 41
CCA	**C**ommon **C**omponent **A**rchitecture, 14
CCM	**C**ORBA **C**omponent **M**odel, 15
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture, 15
CRL	**C**ertificate **R**evocation **L**ist, 41
CTL	**C**omponent **T**emplate **L**ibrary, 14
DSL	**D**omain **S**pecific (programming) **L**anguage [vDKV00], 24
EGEE	**E**nabling **G**rids for **E**-scienc**E** (EGEE) is a project funded by the European Commission's Sixth Framework Programme through Directorate F: Emerging Technologies and Infrastructures, of the Directorate-General for Information Society and Media. See `http://en.wikipedia.org/wiki/EGEE`, 39
EPIC	**E**xplicitly **P**arallel **I**nstruction **C**omputing (EPIC) is a computer instruction set paradigm, where the compiler identifies parallelism. The resulting machine programs are scheduled *in order* onto the processing units. See `http://en.wikipedia.org/wiki/Explicitly_parallel_instruction_computing`, 36
FLOPS	**FL**oating Point **OP**erations per **S**econd, 35
FP	Functional Programming, 10
GenP	**Gen**eric **P**rogramming, 22
glite	A middle-ware for grid computing. Born from the collaborative efforts of more than 80 people in 12 different academic and industrial research centres as part of the EGEE Project See`http://glite.web.cern.ch/glite/`, 39

GLOBUS	The GLOBUS-Toolkit, currently at version 4, is an open source toolkit for building grids. It is provided by the GLOBUS Alliance. See `http://en.wikipedia.org/wiki/Globus_Toolkit`, 39
GRAM	**G**LOBUS **R**esource **A**llocation **M**anager, 43
Grid	A Grid is the name for a globally distributed operating system environment. See Chapter 7., 39
GSI	**G**LOBUS **S**ecurity **I**nfrastructure, 42
GTK	**G**LOBUS **T**ool **K**it, 43
GUI	**G**raphical **U**ser **I**nterface, 7
HPC	**H**igh **P**erformance **C**omputing, 34
Hyper-Threading	Hyper-threading is a technology which provides certain processor resources twice, for accelerating the context switch in multi-threaded applications. See `http://en.wikipedia.org/wiki/Hyperthreading`, 36
IBM	**I**nternational **B**usiness **M**achines, 13
IBM Power	*POWER* is a RISC instruction set architecture designed by IBM. The name is a acronym for **P**erformance **O**ptimisation **W**ith **E**nhanced **RISC**. See `http://en.wikipedia.org/wiki/IBM_Power`, 36
ILP	**I**nstruction-**L**evel **P**arallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. See `http://en.wikipedia.org/wiki/Instruction_level_parallelism`, 36
IP	**I**nternet **P**rotocol, 43
IPC	**I**nter-**P**rocess **C**ommunication is a set of techniques for the exchange of data among two or more threads in one or more processes. See `http://en.wikipedia.org/wiki/Inter-process_communication`, 37
LAM	**L**ocal **A**rea **M**ulti-computer, 38
LDAP	**L**ightweight **D**irectory **A**ccess **P**rotocol, 41
LOC	Lines of Code is a basic code metric [Tha00]., 23
MCMD	**M**ultiple **C**omponent **M**ultiple **D**ata:Parallel programming paradigm having multiple component implementations working on a parallel computer with multiple input data., 38
MIMD	**M**ultiple **I**nstruction **M**ultiple **D**ata in Flynnś taxonomy [Fly72], 36
MISD	**M**ultiple **I**nstruction Single Data in Flynnś taxonomy [Fly72], 36
MOSIX	MOSIX is a management system for Linux clusters and organisational grids that provides a single-system image. The further development of OpenMOSIX was cancelled on 1. March 2008, because new technologies like multi-core and virtualisation making the solution needless., 37
MP	**M**eta **P**rogramming, 21
MPI	**M**essage **P**assing **I**nterface is programming interface for distributed communication, used to program parallel computers. See `http://en.wikipedia.org/wiki/Message_Passing_Interface`, 37

MPMD **M**ultiple **P**rocess **M**ultiple **D**ata or **M**ultiple **P**rogram **M**ultiple **D**ata, see SPMD., 38

MPP **M**assively **P**arallel **P**rocessor, 34

Multi-Core A multi-core CPU (or chip-level multiprocessor, CMP) combines two or more independent processing cores into a single processor package. See `http://en.wikipedia.org/wiki/Multi-core`, 37

NATO The **N**orth **A**tlantic **T**reaty **O**rganisation is a military alliance, established by the signing of the North Atlantic Treaty on April 4, 1949. See `http://en.wikipedia.org/wiki/NATO`, 9

NUMA **N**on-**U**niform **M**emory **A**ccess or **N**on-**U**niform **M**emory **A**rchitecture is a memory design used in multi-processor computers, where the memory access time depends on the location relative to a processor. See `http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access`, 37

OCSP **O**n-line **C**ertificate **S**tatus **P**rotocol, 41

OGSA **O**pen **G**rid **S**ervices **A**rchitecture. It describes an architecture for a service-oriented grid computing environment for business and scientific use, developed within the Global Grid Forum (GGF). See `http://en.wikipedia.org/wiki/Open_Grid_Services_Architecture`, 39

OGSI **O**pen **G**rid **S**ervices **I**nfrastructure: It was published by the Global Grid Forum (GGF) as a proposed recommendation in June 2003. It was intended to provide an infrastructure layer for the Open Grid Services Architecture (OGSA), OGSI takes the statelessness issues (along with others) into account by essentially extending Web services to accommodate grid computing resources that are both transient and stateful. See `http://en.wikipedia.org/wiki/Open_Grid_Services_Infrastructure`, 39

OMG **O**bject **M**anagement **G**roup: It is a consortium originally aimed at setting standards for distributed object-oriented systems, and is now focused on modelling (programs, systems and business processes) and model-based standards in some 20 vertical markets., 15

OOP **O**bject **O**riented **P**rogramming, 15

Opteron The AMD Opteron was the first of AMD's eighth-generation x86 processors, and the first processor to implement the AMD64 (formerly x86-64) instruction set architecture. See `http://en.wikipedia.org/wiki/Opteron`, 36

Pipe In Unix-like computer operating systems, a pipeline (Pipe) is the original software pipeline: a set of processes chained by their standard streams, so that the output of each process (stdout) feeds directly as input (stdin) of the next one. See `http://en.wikipedia.org/wiki/Pipeline_\%28Unix\%29`, 37

PKI **P**ublic **K**ey **I**nfrastructure, 40

QoS **Q**uality **o**f **S**ervice, 44

RMI **R**emote **M**ethod **I**nvocation, 15

RSA **R**ivest **S**hamir **A**dleman: The RSA algorithm was publicly described in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman at MIT; the letters RSA are the initials of their surnames., 40

SCMD **S**ingle **C**omponent **M**ultiple **D**ata: Parallel programming paradigm having a single component implementation working on a parallel computer with multiple input data., 38

SIMD **S**ingle **I**nstruction **M**ultiple **D**ata in Flynnś taxonomy [Fly72], 36

SISD **S**ingle **I**nstruction **S**ingle **D**ata in Flynnś taxonomy [Fly72], 36

SMP **S**ymmetric **M**ulti-**P**rocessor, 34

SOA **S**ervice **O**riented **A**rchitectur: There is no widely agreed definition of service-oriented architecture other than its literal translation that it is an architecture that relies on service-orientation as its fundamental design principle. Service-orientation describes an architecture that uses loosely coupled services to support the requirements of business processes and users. See `http://en.wikipedia.org/wiki/Service-oriented_architecture`, 43

socket An Internet socket (short socket), is a communication end-point unique to a process on a machine communicating on an Internet Protocol-based network, such as the Internet. See `http://en.wikipedia.org/wiki/Internet_socket`, 42

SPMD **S**ingle **P**rocess **M**ultiple **D**ata or **S**ingle **P**rogram **M**ultiple **D**ata is a technique employed to achieve parallelism; it is a subcategory of MIMD. Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster.`http://en.wikipedia.org/wiki/SPMD`, 38

SSE **S**treaming **SIMD** **E**xtensions, originally called is a SIMD (Single Instruction, Multiple Data) instruction set designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD's 3DNow
(which had debuted a year earlier). See `http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions`, 36

SSI **S**ingle **S**ystem **I**mage provides a form of distributed computing in which by using a common interface, multiple networks, distributed databases or servers appear to the user as one system. See `http://en.wikipedia.org/wiki/Single-system_image`, 37

TCP **T**ransmission **C**ontrol **P**rotocol, 43

TOP500 List of the 500 fastest Supercomputers in the world. See `http://www.top500.org`, 34

UNICORE UNICORE (**UN**iform **I**nterface to **CO**mputing **RE**sources) is a Grid computing technology that provides seamless, secure, and intuitive access to distributed Grid resources such as supercomputers or cluster systems and information stored in databases. See `http://en.wikipedia.org/wiki/UNICORE`, 39

VLIW **V**ery **L**ong **I**nstruction **W**ord or VLIW refers to a CPU architecture designed to take advantage of instruction level parallelism (ILP). See `http://en.wikipedia.org/wiki/Very_long_instruction_word`, 36

VO **V**irtual **O**rganisation (VO): In grid computing, a VO is a group of individuals or institutions who share the computing resources of a grid for a common goal. See `http://en.wikipedia.org/wiki/Virtual_organization`, 40

W3C **W**orld **W**ide **W**eb **C**onsortium, 43

Web Service The W3C defines a Web Service as a software system designed to support interoperable Machine to Machine interaction over a network. See `http://en.wikipedia.org/wiki/Webservices`, 43

X.509 Open standard for public key certificates., 40

XEON — The Xeon brand refers to Intel's x86 multiprocessing CPUs targeted at server and workstation computers. See `http://en.wikipedia.org/wiki/XEON`, 36

Bibliography

[AAae06] Benjamin A. Allan, Robert Armstrong, and al. et. A component architecture for high-performance scientific computing. *Intl. J. High-Perf. Computing Appl.*, 20(2):163–202, 2006.

[AAF+01] Gabrielle Allen, David Angulo, Ian Foster, Gerd Lanfermann, Chuang Liu, Thomas Radke, Ed Seidel, and John Shalf. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a Grid environment. *The International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[AGG+99] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing, 1999.

[AGH00] Ken Arnold, James Gosling, and David Holmes. The Java^{TM}programming language, 2000.

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[And04] D. P. Anderson. Boinc: a system for public-resource computing and storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing.*, pages 4–10. IEEE Computer Society, 2004.

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, MA, USA, 1985.

[BCJ] Rajkumar Buyya, Toni Cortes, and Hai Jin. Computing applications single system image (ssi).

[Beh00] Anita Behle. *Wiederverwendung von Softwarekomponenten im Internet*. Deutscher Universitäts-Verlag GmbH, 2000.

[Ber00] Guntram Berti. *Generic Software Components for Scientific Computing*. PhD thesis, TU Cottbus, 2000.

[BFH03] Fran Berman, Geoffrey Fox, and Tony Hey, editors. *Grid Computing*. Wiley, 2003.

[BHB+03] Gerd Beneken, Ulrike Hammerschall, Manfred Broy, Maria Victoria Cengarle, Jan Jürjens, Bernhard Rumpe, and Maurce Schoenmakers. Componentware - State of the Art 2003. In *Proceedings of the CUE Workshop Venedig*, 2003.

[BKS05] R. Berlich, M. Kunze, and K. Schwarz. Grid computing in europe: from research to deployment. In *ACSW Frontiers '05: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*, pages 21–27, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[BL98] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[Boa05] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 2005.

[BP01] Steve Burnett and Stephen Paine. *RSA Security's Official Guide to Cryptography*. RSA Press, 2001.

[Bü07] Boris Bügling. Connecting component architectures. Master's thesis, TU-Braunschweig, 2007.

[Cal96] Franco Callari. Operating Systems Class, 1996.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[Ce06] G. Coulson and B. Schulze etal. Special issue: Middleware for grid computing: A 'possible future'. *Concurrency and Computation: Practice and Experience*, DOI: 10.1002/cpe.1132, 2006.

[CFFK01] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing, 2001.

[CFK+98] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459:62–??, 1998.

[CH01] Bill Councill and George T. Heineman. *Definition of a software component and its elements*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Cle96] Paul C. Clements. A survey of architecture description languages. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[Cle07a] ClearSpeed. *Accelerating high performance and technical computing to deliver more performance-per-watt*, 2007.

[Cle07b] ClearSpeed. *The ClearSpeed Vector Math Library*, 2007.

[Cli98] William D. Clinger. Proper tail recursion and space efficiency. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 174–185, New York, NY, USA, 1998. ACM.

[CLJ+07] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM Press.

[CLR00] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms, 2000.

[Cou98] Bernard Coulange. *Software Reuse*. Springer Verlag, London, 1998.

[CR06] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins (2nd Edition) (The Eclipse Series)*. Addison-Wesley Professional, March 2006.

[Dam05] Kostadin Damevski. Generating bridges between heterogeneous component models. Unpublished contribution at Generative Programming and Component Engineering (GPCE'05), 2005.

[DFH+93] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE '93: Parallel Architectures and Languages Europe*, pages 146–160. Springer-Verlag, Berlin, DE, 1993.

[DR09] Peter J. Denning and Richard D. Riehle. The profession of it is software engineering engineering? *Commun. ACM*, 52(3):24–26, 2009.

[DRR+06] Manuel Díaz, Sergio Romero, Bartolomé Rubio, Enrique Soler, and José M. Troya. Dynamic reconfiguration of scientific components using aspect oriented programming: A case study. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (2)*, volume 4276 of *Lecture Notes in Computer Science*, pages 1351–1360. Springer, 2006.

[DSSS05] Jack Dongarra, Thomas Sterling, Horst Simon, and Erich Strohmaier. High-performance computing: Clusters, constellations, mpps, and future directions. *Computing in Science and Engg.*, 7(2):51–59, 2005.

[DWJ+96] Tzvetan T. Drashansky, Sanjiva Weerawarana, Anupam Joshi, Ranjeewa A. Weerasinghe, and Elias N. Houstis. Software architecture of ubiquitous scientific computing environments for mobile platforms. *Mobile Networks and Applications*, 1(4):421–432, 1996.

[EAH+07] Marc Eaddy, Alfred V. Aho, Weiping Hu, Paddy McDonald, and Julian Burger. Debugging aspect-enabled programs. In *Software Composition*, pages 200–215, 2007.

[EDE03] T. Eidson, J. Dongarra, and V. Eijkhout. Applying aspect-orient programming concepts to a component-based programming model, 2003.

[EF03] Andreas Eberhart and Stefan Fischer. Web services, 2003.

[FKL+99] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, 1999.

[FKT99] Ian Foster, Carl Kesselman, and Steven Tuecke. The grid-blueprint for a new computing infrastructure, 1999.

[Fly72] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 9:948–960, 1972.

[Fos05] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer Verlag, 2005.

[Fox02] Geoffrey C. Fox. *From computational science to internetics: integration of science with computer science*, pages 217–234. Purdue University Press, West Lafayette, IN, USA, 2002.

[FPF99] C.A. Felippa, K.C. Park, and C. Farhat. Partitioned analysis of coupled mechanical systems. Technical report, Dep. of Aero. Eng. Sci. and Cent. for Aero. Struct. at University of Colorado, Boulder, Colorado, USA, 1999. `http::/caswww.colorado.edu/Felippa.d/FelippaHome.d/Publications.d/Report.CU-CAS-99-06.pdf`.

[FPSF06] Niels Fallenbeck, Hans-Joachim Picht, Matthew Smith, and Bernd Freisleben. Xen and the art of cluster scheduling. *vtdc*, 0:4, 2006.

[GAL+03] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Massó, Thomas Radke, Edward Seidel, and John Shalf. The cactus framework and toolkit: Design and applications. In *High Performance Computing for Computational Science - VECPAR 2002*, volume Volume 2565/2003, pages 197–227. Springer Berlin / Heidelberg, 2003.

[GB03] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[GBL04] Madhusudhan Govindaraju, Himanshu Bari, and Michael J. Lewis. Design of Distributed Component Frameworks for Computational Grids. In *Proceedings of the International Conference on Communications in Computing (CIC)*, June 2004.

[GCGS] Dennis Gannon, Kenneth Chiu, Madhusudhan Govindaraju, and Aleksander Slominski. A revised analysis of the open grid services infrastructure.

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GHN04] A. S. Grimshaw, M. A. Humphrey, and A. Natrajan. A philosophical and technical comparison of Legion and Globus. *IBM J. Res. Dev.*, 48(2):233–254, 2004.

[GJWJ07] Brinda Ganesh, Aamer Jaleel, David Wang, and Bruce Jacob. Fully-buffered dimm memory architectures: Understanding mechanisms, overheads and scaling. In *Proc. 13th International Symposium on High Performance Computer Architecture (HPCA 2007)*., 2007.

[GKC+an] Madhusudhan Govindaraju, Sriram Krishnan, Kenneth Chiu, Aleksander Slominski, Dennis Gannon, and Randall Bramley. Merging the CCA Component Model with the OGSI Framework. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 12-15, 2003, Tokyo, Japan.

[Goo00] Gerhard Goos. *Vorlesungen über Informatik Band 1: Grundlagen und funktionales Programmieren*. Springer, 3 edition, Jan 2000.

[Goo01] Gerhard Goos. *Vorlesungen über Informatik Band 2: Objektorientiertes Programmieren und Algorithmen*. Springer, 3 edition, Jan 2001.

[GPF00] J.A. Gonzalez, K.C. Park, and C.A. Felippa. Partitioned formulation of frictional contact problems using localized Lagrange multipliers. *Commun. Numer. Meth. Engng.*, 00:1-6, 2000.

[Gri98] Frank Griffel. *Componentware*. dpunkt.verlag Heidelberg, 1998.

[HDGB99] K. Hameyer, J. Driesen, H. De Gersem, and R. Belmans. The classification of coupled field problems. *IEEE Transactions on Magnetics*, 35, Issue: 3, Part 1, 1999.

[HM06] T. Srisupattarawanit H.G. Matthies, R. Niekamp. Scientific computing with software-components. *IACM Expressions*, 19:27–31, 2006.

[HMR+00] Jerome C. Huck, Dale Morris, Jonathan Ross, Allan D. Knies, Hans Mulder, and Rumi Zahir. Introducing the ia-64 architecture. *IEEE Micro*, 20(5):12–23, 2000.

[HPR05] Pilar Herrero, María S. Pérez, and Víctor Robles, editors. *Scientific Applications of Grid Computing, First International Workshop, SAG 2004, Beijing, China, September 20-24, 2004, Revised Selected and Invited Papers*, volume 3458 of *Lecture Notes in Computer Science*. Springer, 2005.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Hug01] Karlheinz Hug. *Module, Klassen, Verträge*, volume 2. Verlag Vieweg, Wiesbaden, 2001.

[J.K93] J.Kohl. Rfc 1510 - the kerberos network authentication service (v5). Technical report, Digital Equipment Corporation, 1993.

[JLM00] Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors. *Generic Programming, International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*. Springer, 2000.

[JN04] J.P.Lewis and Ulrich Neumann. Performance of versus c++, 2004.

[Jon02] Merijn de Jonge. Source tree composition. In *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *LNCS*. Springer-Verlag, 2002.

[Jü05] Dominik Jürgens. Implementierung einer verteilten Ressourcenverwaltung als Komponente für die Component Template Library. Technical report, Inst. of sci. comp., TU-Braunschweig, 2005.

[Jü06] Dominik Jürgens. Konzept und Implementierung einer Suchanfrage-Sprache für Komponenten-Systeme. Master's thesis, TU Braunschweig, 2006.

[Kal98] Laxikant V. Kale. Programming languages for cse: The state of the art. *IEEE Comput. Sci. Eng.*, 5(2):18–26, 1998.

[KBM02] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Softw. Pract. Exper.*, 32(2):135–164, February 2002.

[KG04] S. Krishnan and D. Gannon. Xcat3: A framework for cca components as ogsa services, 2004.

[KKPR01] Scott R. Kohn, Gary Kumfert, Jeffrey F. Painter, and Calvin J. Ribbens. Divorcing language dependencies from a scientific software library. In *PPSC*, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KNS+04] K. Kalapriya, S. K. Nandy, Deepti Srinivasan, R. Uma Maheshwari, and V. Satish. A framework for resource discovery in pervasive computing for mobile aware task execution. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 70–77, New York, NY, USA, 2004. ACM.

[KS05] H. Kuchen and J. Striegnitz. Features from functional programming for a c++ skeleton library: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(7-8):739–756, 2005.

[KTB05] Nitin V. Kanaskar, Umit Topaloglu, and Coskun Bayrak. Globus security model for grid environment. *SIGSOFT Softw. Eng. Notes*, 30(6):1–9, 2005.

[Lae04] J. Walter Larson and al. et. Components, the common component architecture, and the climate/weather/ocean community. In *84th American Meteorological Society Annual Meeting*, Seattle, Washington, 2004. American Meteorological Society.

[LG86] Barbara Liskov and John Guttag. *Abstraction and specification in program development*. MIT Press, Cambridge, MA, USA, 1986.

[Loo01] Peter Loos. *GoTo COM*. Addison-Wesley, 2001.

[LSNA97] Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz, and Franz Achermann. Towards a formal composition language. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of ESEC '97 Workshop on Foundations of Component-Based Systems*, pages 178–187, Zurich, 1997.

[LW04] B.P. Lamichhane and B.I. Wohlmuth. Mortar finite elements with dual lagrange multipliers: Some applications. In R. Kornhuber, R. Hoppe, J. Pèriaux, O. Pironneau, O. Widlund, and J. Xu, editors, *Fifteenth International Conference on Domain Decomposition Methods*, pages 319–326, 2004.

[Mae06] Lois Curfman McInnes and al. et. Parallel PDE-based simulations using the Common Component Architecture. In Are Magnus Bruaset and Aslak Tveito, editors, *Numerical Solution of PDEs on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering (LNCSE)*, pages 327–384. Springer-Verlag, 2006. invited chapter, also Argonne National Laboratory technical report ANL/MCS-P1179-0704.

[McI68] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[MNS06] Hermann G. Matthies, Rainer Niekamp, and Jan Steindorf. Algorithms for strong coupling procedures. *Comp. Meth. in Appl. Mech. and Eng.*, 195:2028–2049, 2006.

[Mol06] Ethan Mollick. Establishing moore's law. *IEEE Annals of the History of Computing*, 28(3):62–75, 2006.

[MSM05] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Pearson Education, Inc., 2005.

[MSZ06] Jens-Michael Milke, Michael Schiffers, and Wolfgang Ziegler. Virtuelle organisationen in grids. charakterisierung und management, 2006. in German only.

[Neu00] John Von Neumann. *The Computer and the Brain*. Yale University Press, New Haven, CT, USA, 2000. Foreword By-Paul M. Churchland and Preface By-Klara Von Neumann.

[NHK+] J. Nieplocha, R.J. Harrison, M.K. Kumar, B. Palmer, V. Tipparaju, and H. Trease. Combining distributed and shared memory models: Approach and evolution of the global arrays toolkit.

[Nie07a] Rainer Niekamp. Ctl manual for linux/unix, 2007.

[Nie07b] Rainer Niekamp. Ctl project web, 2007.

[NKcS03] Sivaramakrishnan Narayanan, Tahsin M. Kurç, Ümit V. Çatalyürek, and Joel H. Saltz. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 13(2):245–271, 2003.

[NR69] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969.

[NS02] Zsolt Németh and Vaidy S. Sunderam. A comparison of conventional distributed computing environments and computational grids. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, pages 729–738, London, UK, 2002. Springer-Verlag.

[NSW03] Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors. *Grid Resource Management*. Kluwer Academic Publishers, 2003.

[NVI07a] NVIDIA. *CUDA CUBLAS Library*, 2007.

[NVI07b] NVIDIA. *CUDA CUFFT Library*, 2007.

[NVI07c] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*, 2007.

[OMG06] Object Management Group OMG. Corba component model, v4.0 (formal/2006-04-01), 2006.

[Par] Parallel Programming Laboratory, University of Illinois at Urbana-Champaign, http://charm.cs.uiuc.edu/manuals/html/charm++/manual.html. *The Charm++ Programming Language Manual*, version 5.8 (release 1) edition.

[Par06] S.G. Parker. A component-based architecture for parallel multi-physics pde sim ulation. *Future Generation Computer Systems*, 22(1-2):204–216, 2006.

[PF00] K.C. Park and C.A. Felippa. A variational principle for the formulation of partitioned structural systems. *Internat. J. Numer. Methods Eng.*, 47:395–418, 2000.

[PFG00] K.C. Park, C.A. Felippa, and U.A. Gumaste. A localized version of the method of Lagrange multipliers and its applications. *Computational Mechanics*, 24(6):476–490, 2000.

[PFO04] K.C. Park, C.A. Felippa, and R. Ohayon. Reduced-order partitioned modeling of coupled systems: Formulation and computational algorithms. In *Proc. NATO-ARW Workshop on Multi-physics and Multi-scale Computer Models in Non-linear Analysis and Optimal Design of Engineering Structures Under Extreme Conditions, Bled, Slovenia*, 2004.

[PFR01] K.C. Park, C.A. Felippa, and G. Rebel. Interfacing nonmatching FEM meshes: The zero moment rule. Technical report, Dept. of Aero. Eng. Sci. and Cent. for Aero. Struct. at University of Colorado, Boulder, Colorado, USA, 2001. `http://cas.colorado.edu/main.d/PDF.d/01-01.pdf`.

[PFR02] K.C. Park, C.A. Felippa, and G. Rebel. A simple algorithm for localized construction of non-matching structural interfaces. *Int. J. Numer. Meth. Eng.*, 53:2117–2142, 2002.

[Pie99] Claudia Piemont. *Komponenten in JAVA*. dpunkt Verlag, 1999.

[PT97] William H. Press and Saul A. Teukolsky. Numerical recipes: Does this paradigm have a future?, 1997.

[QRH] J. Qiang, R. Ryne, and S. Habib. Implementation of object-oriented design with fortran language in beam dynamics studies.

[Rom99] Mathilde Romberg. The unicore architecture: Seamless access to distributed resources. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 44, Washington, DC, USA, 1999. IEEE Computer Society.

[Rus04] Michael Russell. Quality busters: Don't violate the principle of locality, 2004.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Symposium on Principles of Distributed Computing*, pages 1–7, 1996.

[SBG96] Barry F. Smith, Petter E. Bjørstad, and William Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[Sch99] J.-G. Schneider. *Components, scripts, and glue: A conceptual framework for software composition, Ph.D. Thesis*. PhD thesis, University of Bern, October 1999.

[Sch03] Hartmut Schwandt. *Parallele Numerik*. Teubner Verlag, Wiesbaden, 2003.

[SGK07] S. V. Sharma, G. Gopalakrishnan, and R. M. Kirby. A survey of mpi related debuggers and tools. Technical Report UUCS-07-015, University of Utah, School of Computing, 2007.

[SL96] Jean-Guy Schneider and Markus Lumpe. Modelling objects in PICT. Technical Report IAM-96-004, University of Bern, 1996.

[SL03] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[Slo05] Joseph D. Sloan. *High Performance Linux Clusters*. O'Reilly Media, 2005.

[SMH03] T. Sloane, M. Mernik, and J. Heering. When and how to develop domain-specific languages, 2003.

[SNS88] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 191–202, Berkeley, CA, 1988. USENIX Association.

[Som04] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[Spw] Sean Wentworth Spw. Performance evaluation: Java vs c++.

[Sria] Madhusudhan Govindaraju Sriram. Merging the cca component model with the ogsi framework.

[Srib] Madhusudhan Govindaraju Sriram. Xcat 2.0: Design and implementation of component based web services.

[Sri97] Prashant Sridharan. *Advanced Java networking*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[SS06] Stefan Gruner Saša Subotić, Judith Bishop. Aspect-oriented programming for a distributed framework. `http://polelo.cs.up.ac.za/papers/SuboticAspects.pdf`, 2006.

[ST07] Gernot Starke and Stefan Tilkov, editors. *SOA Expertenwissen–Methoden, Konzepte und Praxis serviceorientierter Architekturen*. dpunkt-Verlag, 2007.

[Ste04] Jan Steindorf. *Partitionierte Verfahren fuer Probleme der Fluid-Struktur Wechselwirkung*. Facbereich fuer Mathematik und Informatik, 2004.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Szy97] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[Tea02] The BlueGene/L Team. An overview of the bluegene/l supercomputer, 2002.

[Tha00] Georg E. Thaller. *Software-Metriken einsetzen - bewerten - messen*. Verlag Technik, 2000.

[TTL02] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[TvS03] Andrew Tannenbaum and Marten van Steen. Verteilte Systeme, 2003.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[Vel06] Todd L. Veldhuizen. Tradeoffs in metaprogramming. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 150–159, New York, NY, USA, 2006. ACM.

[Wät04] Dietmar Wätjen. *Kryptographie*. Spektrum Akademischer Verl., 2004.

[Weg97] Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, 1997.

[WG06] D.J. Worth and C. Greenough. A survey of software tools for computational science. Technical Report 11, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, 2006. http://epubs.cclrc.ac.uk/work-details?w=39927.

[WGC+04] T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 303–310, Budapest, Hungary, September 2004.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition (Paperback)*. Addison-Wesley, 2003.

[Woh01] Barbara Wohlmuth. *Discretization Methods and Iterative Solvers Based on Domain Decomposition*. Springer Verlag, Berlin, Heidelberg,, 2001.

[WOZ91] Bruce W. Weide, William F. Ogden, and Stuart H. Zweben. *Reusable software components*, pages 1–65. Academic Press Professional, Inc., San Diego, CA, USA, 1991.

[WRH02] Eric Whipple, Rick Ross, and Nick Hodges. *Softwareentwicklung mit Kylix*. Software & Support Verlag, 2002.

[Wri02] Peter Wriggers. *Computational Contact Mechanics*. Wiley, Chichester, 2002.

[WSH99] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[ZMN05] Fen Zhu, Matt W. Mutka, and Lionel M. Ni. Service discovery in pervasive computing environments. *IEEE Pervasive Computing*, 4(4):81–90, 2005.

[ZS06] Wolf Zimmermann and Michael Schaarschmidt. Automatic checking of component protocols in component-based systems. In *Software Composition*, pages 1–17, 2006.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2005-08

2005-08	B. Florentz, M. Huhn	A Metamodel for Architecture Evaluation
2006-01	T. Klein, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshops MBEES 2006: Modellbasierte Entwicklung eingebetteter Systeme
2006-02	T. Mücke, B. Florentz, C. Diefer	Generating Interpreters from Elementary Syntax and Semantics Descriptions
2006-03	B. Gajanovic, B. Rumpe	Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen
2006-04	H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel	Handbuch zu MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen
2007-01	M. Conrad, H. Giese, B. Rumpe, B. Schätz (Hrsg.)	Tagungsband Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme III
2007-02	J. Rang	Design of DIRK schemes for solving the Navier-Stokes-equations
2007-03	B. Bügling, M. Krosche	Coupling the CTL and MATLAB
2007-04	C. Knieke, M. Huhn	Executable Requirements Specification: An Extension for UML 2 Activity Diagrams
2008-01	T. Klein, B. Rumpe (Hrsg.)	Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen, Tagungsband
2008-02	H. Giese, M. Huhn, U. Nickel, B. Schätz (Hrsg.)	Tagungsband des Dagstuhl-Workshopss MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV
2008-03	R. van Glabbeek, U. Goltz, J.-W. Schicke	Symmetric and Asymmetric Asynchronous Interaction
2008-04	M. V. Cengarle, H. Grönniger B. Rumpe	System Model Semantics of Statecharts
2008-05	M. V. Cengarle, H. Grönniger B. Rumpe	System Model Semantics of Class Diagrams
2008-06	M. Broy, M. V. Cengarle, H. Grönniger B. Rumpe	Modular Description of a Comprehensive Semantics Model for the UML (Version 2.0)
2008-07	C. Basarke, C. Berger, K. Berger, K. Cornelsen, M. Doering J. Effertz, T. Form, T. Gülke, F. Graefe, P. Hecker, K. Homeier F. Klose, C. Lipski, M. Magnor, J. Morgenroth, T. Nothdurft, S. Ohl, F. Rauskolb, B. Rumpe, W. Schumacher, J. Wille, L. Wolf	2007 DARPA Urban Challenge Team CarOLO - Technical Paper
2008-08	B. Rosic	A Review of the Computational Stochastic Elastoplasticity
2008-09	B. N. Khoromskij, A. Litvinenko, H. G. Matthies	Application of Hierarchical Matrices for Computing the Karhunen-Loeve Expansion
2008-10	R. van Glabbeek, U. Goltz, J.-W. Schicke	On Synchronous and Asynchronous Interaction in Distributed Systems
2009-01	H. Giese, M. Huhn, U. Nickel, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme V
2009-02	D. Jürgens	Survey on Software Engineering for Scientific Applications: Reuseable Software, Grid Computing and Application